

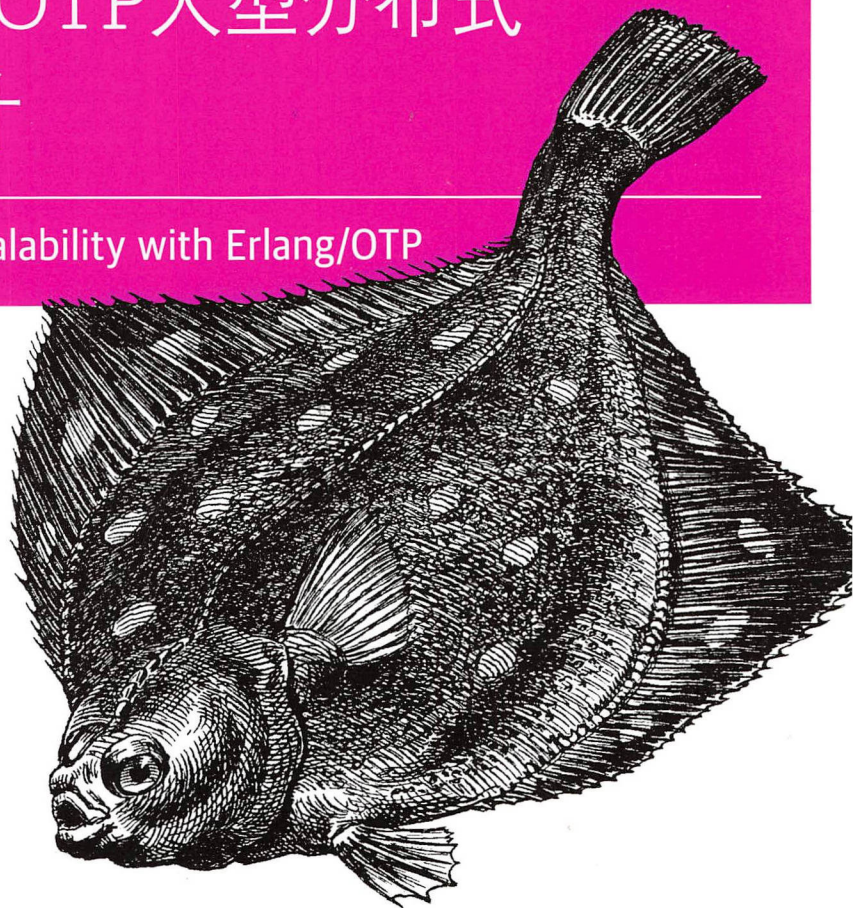
版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

高伸缩性系统

Erlang/OTP大型分布式 容错设计

Designing for Scalability with Erlang/OTP



[瑞典] Francesco Cesarini [美] Steve Vinoski 著
林建入 译

 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



O'REILLY®

高伸缩性系统

Erlang/OTP大型分布式容错设计

Designing for Scalability with Erlang/OTP

[瑞典] Francesco Cesarini [美] Steve Vinoski 著
林建入 译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING



内 容 简 介

这是一本罕见的站在核心设计者而非普通开发者角度介绍 Erlang/OTP 系统的权威书籍。两位作者均是深耕分布式计算领域超过20年的专家。本书内容兼具深度与广度，不仅带领读者通过一步步实践的方式深入剖析了 Erlang/OTP 中各类核心进程的行为模式的设计原理，并且还介绍了特殊进程、自定义行为模式、发行包制作等高级主题。除此之外，本书还用了大量篇幅向读者介绍了 Erlang/OTP 系统中的设计原则、构建分布式系统的方法，以及在此基础上实现容错和规模伸缩所需了解的相关知识。

对于任何一位渴望基于 Erlang/OTP 构建出商业级的分布式、高伸缩性、容错型系统的开发者，本书都是不容错过的经典之作。

© 2016 by Francesco Cesarini and Steve Vinoski.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2018. Authorized translation of the English edition, 2016 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media, Inc.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2017-5971

图书在版编目（CIP）数据

高伸缩性系统：Erlang/OTP大型分布式容错设计 /（瑞典）弗朗西斯科·切萨里尼（Francesco Cesarini），（美）史蒂夫·温斯基（Steve Vinoski）著；林建入译. —北京：电子工业出版社，2018.6

书名原文：Designing for Scalability with Erlang/OTP

ISBN 978-7-121-33747-5

I. ①高… II. ①弗… ②史… ③林… III. ①程序语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2018）第036188号

策划编辑：张春雨

责任编辑：刘 舫

封面设计：Karen Montgomery 张 健

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱

邮编：100036

开 本：787×980 1/16

印张：29

字数：636千字

版 次：2018年6月第1版

印 次：2018年6月第1次印刷

定 价：115.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zltz@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。



O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始, O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来, 而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者, O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”; 创建第一个商业网站 (GNN); 组织了影响深远的开放源代码峰会, 以至于开源软件运动以此命名; 创立了 Make 杂志, 从而成为 DIY 革命的主要先锋; 公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖, 共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择, O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程, 每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人, 他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了: ‘如果你在路上遇到岔路口, 走小路(岔路)。”回顾过去 Tim 似乎每一次都选择了小路, 而且有几次都是一闪即逝的机会, 尽管大路也不错。”

——Linux Journal



一封感谢信

感谢 Alison、Peter 和 Bump 宝宝的耐心和支持。

——*Francesco*

感谢 Dooley、Ed 教授了我做法；感谢 Cindy、Ryan、Erin、
Andrew 和 Jake 告诉了我原因。

——*Steve*

感谢 Joe、Mike 和 Robert 当年为拨通那个电话所发明的这一切。

——*Francesco & Steve*



推荐序一

随着 IT 系统的蓬勃发展，万台级别的机器相互协作组成一个复杂的业务系统已经不再罕见。特别是云计算的普及，使得基础计算资源的获取变得便捷高效，在几个小时内计算资源被申请，各种各样用途的节点被部署，节点被有机互联协作，系统被监控和运维，业务系统被迅速扩容，发现的问题被迅速解决。

而实现这样复杂业务的分布式系统一直是一个很热很大的话题，系统的鲁棒性、可伸缩性、高可用性等每个话题单拎出来都可以讲几天几夜。领域固有的复杂性让很多人望而生畏，更糟糕的是现实对于问题的解法各不相同，很难找到系统讲解这些知识的著作来解释 Why、How，更别提最佳实践了。

2005 年我在开发 VOIP 系统的时候，偶然发现我接手的分布式系统，虽然是用 C 语言编写的，但很奇怪地用了进程、消息传递、状态机的架构，而不是熟悉的线程、并发、switch/case 传统解法。当时我负责这套架构的基础设施，为了达到可伸缩、稳定，做得异常痛苦，后来了解到这套架构来自爱立信。2007 年，从 CSDN 的一篇介绍文章开始，爱立信开源的 Erlang 为人所知，而它号称的系统几个 9 的可靠性一直是我不理解的，当我花了很多时间深入了解了 Erlang 语言、OTP 框架、VM 实现后突然豁然开朗，疑惑终于被解开了。

Erlang 语言有自己的哲学和世界观，其试图构建一个和人类系统很像的系统：每个人都是一个进程，人之间是有边界的，人是有组织关系的，人通过消息来协作，每个人都会犯错，每个人的表现都需要被监督。围绕着这个哲学，Erlang 对现实世界描述的进程、消息、协作被具象化，函数式不变语法保证了实体的独立性。它的 VM 被设计成一个类 UNIX 的平台可移植虚拟机系统，对于 CPU、IO、内存的能效管理接近线性的效率，在节点间屏蔽了消息传递的各种复杂性。特别难能可贵的是，自省第一天就被充分植入系统。

这还不够，毕竟会造车和会开车是两件事情，Erlang 用 OTP 框架来强制对分布式的最佳



实践。分布式系统应该如何被设计，如何实现鲁棒性、高可用性，如何伸缩都有精心的引导，只要遵循这些 OTP 行为，很容易达到你的设计目标。

复杂应用生命周期里会有大量部署、运维、监控、问题排查、升级、下线等烦琐的事情要做，特别是系统规模大了以后，业务平滑变成很大的挑战。OTP 很体贴地来救驾，它提供了各种各样的组件让生命周期管理不那么痛苦，可圈可点的是，成体系的平滑热升级能力是从头到脚设计的。作为一个在电信行业里经受二十多年挑战的系统，系统的稳定可靠一直是被刻意追求的，也是值得信赖的。

作者 Francesco 从 R1 就开始参与系统的设计和实现，是一个资深的开发者，对于系统的设计哲学理解深入骨髓，同时他非常热心地创办了 Erlang Solutions，通过各种会议和书籍教授和帮助用户，积累了丰富的经验。而 Steve 在分布式系统上同样沉浸多年，也是资深 Erlang 用户，开发了大量系统。在这本书中，他们把自己的经验、心得倾囊而出，特别是对于大型复杂系统的设计、权衡和妥协的实践，可以帮助读者将知识迅速转化成生产力。

祝学得开心！

褚霸

阿里云研究员



推荐序二

“多少年过去了”，每次提起 Erlang，我都会如此感慨，也都会想起这些年跟 Erlang 有关的一幕幕场景。

想起第一次使用 Erlang，惊叹于跟面向对象完全不一样的函数式编程范式；想起第一个真正的工作项目，用 Erlang 调用 C++ 搜索库实现的智能问答机器人。也更会想起，邀请 Joe Armstrong 来公司交流的场景，想起请教 Steve Vinoski 关于调试修改 Yaws 的邮件，想起遇到问题时，翻霸爷的博客、看 Erlang Factory 各种分享的那些时刻。

虽然比起工作中遇到的各种各样的挑战，这样的场景并不算多，但也正因为如此，才使得它们在时光里显得格外明亮。

这种感觉，相信很多朋友都深有体会，你对 Erlang 有多少喜爱，就有多少叹息。毕竟它作为一门小众语言存在了太久，而且在可预见的未来一段时间，似乎也不会有太大的改变。

而前面提到的这些人，和未提到的很多社区成员一起，仍然在数年甚至数十年如一日地像灯塔般照耀着这个社区，为这门语言的发展做着自己的贡献。多少年来，无数程序员在他们的感召下，学习及体验并发、容错、函数式编程，又在他们的帮助下，实现了一个又一个永不停机的系统。

“多少年过去了”，也是 Francesco 在写完这本书之后的感慨。好在这份持续贡献的心，终究还是让这本书诞生于世。又借助于本书编辑和译者的辛勤努力，国内的开发者得以看到它的中文版本。

他们都是值得被铭记和感谢的。

也因此，当我看到本书的两位作者是 Francesco 和 Steve 时，我并不意外，而且内心的感觉是，这本书由他们写再好不过了。Steve 不用多说，做 Erlang 许多年，为 OTP、Yaws 等众多知名开源项目做出了重要贡献，之前他是公认的 CORBA 宗师，在分布式系统领



域拥有丰富的实践经验。而 Francesco 是 Erlang Solutions 的创始人和技术总监，后者不仅一直举办 Erlang Factory Conference，还一直在 Erlang 开发和培训方面持续耕耘。

重要的是，这两位作者都有足够的毅力写作，也愿意付出多年的精力来完成这样一本事无巨细的著作。因为 Erlang/OTP 涉及的内容非常广泛，它不仅包含 Erlang 语言的各种特色用法，也有针对不同场景的典型设计模式，还包括在分布式系统设计和实现过程中用到的诸多理论。尤其是后面两者，都是需要一定的基础才能理解透彻从而高效使用的。

在过去的三年半中，我们做了中国最大的即时通信 PaaS 平台，也是整个团队真正大量实践 Erlang 的时期。千万级同时在线的用户，不仅意味着大量的长链接，有缓存和存储组件的普遍使用，也有高并发的内部系统间调用。而我们始终坚持使用最简单的 Erlang，坚持使用 OTP 原生而不是第三方的组件，坚持通过测试和实践检验而不是难辨真假的传说来选型。

事实证明，这样的坚持是值得的，同时也证明了 Erlang/OTP 在现代互联网架构下是经得住考验的。我们做到了核心系统整体 99.99% 的可用性，这一方面要归功于容错监督机制的良好运行，更是因为使用了热更新机制的发行包工具，让我们可以真正快速迭代，系统的升级和回滚都可以在秒级完成。

我们用到的大多数工具在本书中都有专门的章节讲述，所以对于本书的常规用法，我的建议是通读后留在案头作为参考。不求全部记住，只求用的时候能想起来就好。

对于 Erlang 的初学者来讲，在学会语言后，下一步就是实现业务系统，OTP 无疑是一把现成的武器。事实上，它不仅是现成的武器，更是一把好用的武器，对于大多数资深的开发者来讲都是足够好用的。

一些有抱负的开发者可能会发现，这些工具的实现原理十分简单，手写一个并不是什么难事。这里我也多做一个提醒，魔鬼都在细节里。当你把所有情况都考虑周全，很大可能写出来的跟这个差不多。所以，相信 OTP 的高效，用它，用好它，做更好玩的事情吧，让这个社区继续独立有趣地走下去。

多少年过去了，我依然可以回忆起刚开始用 Erlang/OTP 的感觉，一如古龙书中所述：就像黑暗中忽然有了光，月光，圆月。

一乐（梁宇鹏）

块聊链创始人



译者序

这是一本值得每个 Erlang 程序员阅读的好书，因为它深入透彻地讲解了 Erlang 程序员进阶过程中最为关键的一环——对 OTP 框架的深入理解。

众所周知，与一些火热的流行语言相比，Erlang 书籍一直以来数量不多，并且其中大多数以介绍入门级内容为主，意在引起读者的兴趣。尽管这些书籍也各具特色，不少堪称佳作，但对于真正需要从事 Erlang 进行开发的程序员来说，仅了解基本内容是远远不够的。因此长期以来，要想进一步学习，就需要自己在 Erlang 文档中摸索。客观地说，Erlang 拥有非常完善的文档，并且其源代码很容易读懂，因此只要有好奇心，你可以深入了解任何你感兴趣的细节。但是，了解细节是一回事，了解细节背后的设计动机又是另一回事。从这个角度来看，文档与源代码虽然将核心机制毫无保留地呈现在我们面前，但仍然有所欠缺。欠缺的是一条线索，一条能够贯穿系统设计中重大决策背后动机的线索。而本书的出版，终于补上了这缺失的一环。

我想强调，本书对于 OTP 的讲解，并非局限于讲解其“用法”——如果真是如此，那么阅读文档便足够——而是更注重其“原理”。此原理既是指其工作流程，更是指其设计动机。正因为如此，本书的内容才显得独特而可贵。具体来说，本书的前半部分，在作者的带领下，读者可跟随其指导重新实现 OTP 中核心的构成要件。这一过程并非平庸的代码罗列然后逐句解读，而是首先从背景出发，遵循一定的设计理念，先带领读者设计出一个小型的模型，其虽然看似简陋，但已能够实现基本功能，然后进一步指出其不足，并将其改进为符合 OTP 理念的实现。与 OTP 内部真正的实现相比，显然读者的实现依然是简陋的，但是却深刻地反映了真实系统运作时的核心原理。倘若读者有心，能够认真跟着作者的指点完成整个过程，那么不仅能够轻松理解这些 OTP 框架中核心构件的使用方法，知其然；并且能够明白其背后的工作原理，知其所以然。

完整介绍完 OTP 后，本书的篇幅已过大半。我想，本书内容即使自此戛然而止，也不愧列入经典之列了。但两位作者 Francesco 和 Steve 却选择更进一步，带领读者探索更深的主题。

于是在第 11 章，我们不仅可学习到 OTP 系统的核心设计原则，并且还跟着作者一步步手工完成了 OTP 发行包 (Release) 的制作。这一章我特别喜欢。因为我和很多读者一样，能使用 rebar3 之类的工具自动完成发行包制作，但对其中的过程却不甚清楚。作者为什么要花费很长的篇幅介绍如何手工制作发行包呢？因为通过这个过程，读者能够深入理解 Erlang 系统的构成及其启动过程。如果不了解这些内容，就无法理解和应对一些比较棘手的启动阶段的问题，同时也丧失了利用这个过程完成一些定制化能力的机会。并且，理解这些内容，对于那些想进一步探索 Erlang 核心机制的硬核程序员来说，也极有帮助。这一章我个人认为是本书中特别重要的一章，并且实践性极强，建议读者跟随作者的指引一步步完成实验。

而第 12 章，更进一步，向读者介绍了如何进行系统升级。我相信很多人都听过 Erlang 支持热更新，但是对它的认识仅限于模块级的热更新。你想知道如何升级 application，甚至升级整个 Erlang 虚拟机吗？事实上一点也不难，作者将告诉你最佳做法，你不用担心升级时 application 间的依赖、数据库模式变化等诸多问题。一切答案都在本书中。剩下的第 13 章到第 16 章同样不容错过。分别介绍了分布式系统架构方案、容错性设计、规模伸缩方法，以及监视与抢救性支持等内容。

每一章都很精彩，我很想向读者一一介绍，但我想更好的做法是让读者自己去领略吧。在这篇译者序里我就不“剧透”了。

交流与反馈

我在 GitHub 上建立了一个项目，如果你希望与我或者其他读者交流，这是一个不错的方式。其中还整理了一些与本书相关的资料（代码、勘误和相关文档等）链接，方便查阅。这个项目会长期维护，欢迎随时来访，共同交流。

<https://github.com/Jianru-Lin/scalabilitywitherlangotp>

回顾与感谢

作为一篇译者序来说，感谢部分一般的做法是优雅而礼貌的寥寥数语带过即可。少则三两句，多则一两段足矣。先是感谢编辑，然后是感谢家人和朋友。这样做或许没有问题，但我仔细想想倘若多年后自己翻起本书，却看不到自己当时真正想说的话，会很遗憾吧。所以还是想把自己真实的想法写下来。

两年前张春雨老师找到我，问我有没有兴趣翻译一本 Erlang 的书。我当然开心地答应了，因为我很喜欢 Erlang。但由于个人业余时间有限，最终花了两年时间才完成。这期间并

非一帆风顺，有很多波折，主要是我个人工作环境发生变化，业余时间有时候很紧张，而且身体有一段时间也有一些不适，综合各种因素导致翻译的进度时好时坏。拖稿也从偶尔有之到家常便饭，编辑从时不时查阅进度，到不间断地催稿。

刚开始编辑是客气地催稿，我则客气地回应。但是次数多了，有时候确实给编辑着急得不行：“这都周三了，说好上周末交的呀？”“最迟这周五，不能再拖了！”我也“压力山大”，只能赶紧抽时间处理，有时候一再拖延，真的是很不好意思回编辑的微信了。于是有“林老师，干什么去了？弄完了吗？”“稿子什么时候能给，急死了！”刚开始是张春雨老师催，后来和刘舫老师两位一起交替催，催得厉害了，有一天，刘舫编辑自己笑着打趣说：“天天追杀啊”。大家都笑了，我也笑了。

我记得很多次，我白天工作忙抽不出时间，只能深夜处理。于是把稿子发给刘舫编辑的时候，已经是凌晨四五点了。可是令我惊奇的是，经常很快就收到了回复。聊了两句后，我准备休息，心里嘀咕着，刘舫老师现在还醒着？深夜交稿尚且如此，周末和节假日更别说了。想想自己尚且有周末休息，可是编辑却一直处于工作状态，顿时觉得自己的辛苦其实和他们还是比不了的。所以对于催稿这件事，也不能说是编辑施压译者，其实编辑同样不容易。

说起来还闹过一个笑话，因为我偶尔会去北京，于是也想见见张春雨老师和刘舫老师。于是有一次就和刘舫老师提起见面吃饭的事，当时文字交谈过程中感觉刘舫老师似乎不太方便。后来才知道原来刘舫老师是女编辑！我和人家沟通了一段时间连对方性别都没搞清楚，真是十分尴尬。但是这也不能完全怪我，因为每次我发的稿件刘舫老师总是细细阅读后给出很多专业的修改意见，让我觉得很厉害，潜移默化习惯性地以为是男同胞。怪只能怪自己有错误的刻板印象。而且后来发现很多技术书籍的编辑都是女性，心里就更惊讶和钦佩了。

其实张春雨老师联系我之前，我早就知道他了，因为我读过的不少优秀引进书籍的策划编辑都是他，我书架上的《游戏引擎架构》和《Clojure 编程》就是（后者的责任编辑还是刘舫老师），这些书都属高水准作品。而其中每一本的译者序里都有“感谢张春雨老师”的话语，这就是为什么我对他有印象的原因。提到这一点，张春雨老师幽默地开玩笑说“呵呵，他们没有感谢我，是我自己加进去的”，把我和刘舫老师都逗乐了。

好吧，不管怎么说本书终于翻译完了。我写了这么长的一段，其实只是希望下次您看到书籍时，不仅要注意到作者和译者的名字，也应当留意编辑们的名字。作为译者，我可以留下一些文字。但作为编辑，就很少让读者意识到他们辛苦的付出了。所以，感谢张春雨老师和刘舫老师，你们辛苦了。

另外，要特别感谢我的妻子，是你一直在催稿，催得比编辑还紧（二位编辑万万没想到吧？

其实你们有一个不花钱的手下天天跟着我，我逃得过你们却逃不过她)，所以现在终于完成了，而不是再多三个月。当然，当我完成这一切时，你比我还要开心。说起来我还欠你一条比目鱼，咱们说好了完成这本书后就买一条尝尝的。你还说，很期待书印刷出来后，捧在手里的感觉，你要看看我在里面是怎么感谢你的。仔细想想这些年我成功做到的每一件事情背后其实都离不开你的支持，但我觉得这还不够，我们还要一起再翻译更多书，一起完成更多想做的事，一起去更多想去的地方。我写下这些文字的时候你就躺在我身后，不亦乐乎地玩着手机。我没有让你看到我写的内容，不过我猜你看到这段的时候一定会高兴的。因为我也。

最后，感谢我的父母和家人，尤其是保慈林女士、保慈芬女士、张绍光先生，是你们令我接受好的教育，并教会我勇敢追求渴望的人生。而我的丈母娘在我工作繁忙期间，在生活上给了我很多关照，减轻了我的很多负担，为我节约了很多时间，对顺利交稿功不可没，我心里十分感激。

说得有些冗长，深感抱歉，但这些都是我的真实想法。因为我想即使再过很多年，读起这段文字还是会很快乐。我很满足。

林建入

2018年5月6日深夜于海口

目录

序言	xxii
第1章 概述	1
定义问题	2
OTP	4
Erlang	6
工具和库	7
系统设计原则	9
Erlang 节点	10
分布式、基础设施、多核	11
总结	12
通过本书你将学到什么	13
第2章 Erlang 简介	18
递归与模式匹配	18
受函数式的影响	22
玩转匿名函数	22
列表推导：生成与测试	23
进程与消息传递	25
不怕出错！	30
用于监督的链接与监视器	31
链接	31
监视器	33
记录	34

映射组	37
宏	38
模块升级	39
ETS：Erlang 元素存储	41
分布式 Erlang	44
命名与通信	45
节点间的连接与可见性	45
总结	47
接下来是什么	47
第 3 章 行为模式	49
进程的骨架	49
设计模式	52
回调模块	53
抽取出通用的行为模式	56
启动 server	57
client 函数	60
server 循环	62
server 内部函数	64
通用服务器	65
消息传递：冰山之下	68
总结	71
接下来是什么	72
第 4 章 通用型服务器	73
gen_server	73
behavior 指令	74
启动一个 server	75
消息传递	77
同步式消息传递	78
异步式消息传递	79
其他消息	81
未处理的消息	82
同步客户端	83

终止	84
调用超时	86
死锁	89
通用型 server 的超时问题	90
使 behavior 休眠	92
全局化	92
链接 behavior	94
总结	94
接下来是什么	95
第 5 章 深入控制 OTP 行为模式	96
sys 模块	96
追踪与记录	96
系统消息	98
你自己的追踪函数	98
统计信息和当前状态	99
sys 模块总结	102
分裂时的可选项	103
内存管理与垃圾回收	104
分裂时应该避免使用的可选项	108
超时	109
总结	109
接下来是什么	109
第 6 章 有限状态机	110
Erlang 风格的有限状态机	111
Coffee FSM	112
硬件桩	114
Erlang 版咖啡机	114
gen_fsm	118
一个基于行为模式的例子	119
启动 FSM	119
发送事件	123
终止	132
总结	133

亲力亲为.....	134
电话控制器.....	134
让我们测试一下.....	136
接下来是什么	138
第 7 章 事件处理器	139
事件	139
通用事件管理器 / 处理器.....	141
启动 / 停止事件管理器	141
添加事件处理器	142
删除事件处理器	144
发送同步的或异步的事件	145
获取数据	148
对错误以及无效返回值的处理.....	150
交换事件处理器	152
融会贯通	154
SASL 警报处理器	157
总结	159
接下来是什么	159
第 8 章 监督者	160
监督树.....	161
OTP 监督者.....	165
监督者行为模式	166
启动监督者.....	166
监督者规格.....	169
动态子进程.....	176
非 OTP 兼容进程	184
可伸缩性和短期进程.....	186
确定性同步启动	187
测试你的监督策略	188
与传统方法相比又如何.....	190
总结	190
接下来是什么	191

第 9 章 OTP application	192
OTP application 是如何运行的	193
OTP application 的结构.....	194
回调模块	198
启动和停止 application	198
application 资源文件.....	202
基站控制器的 application 文件	204
启动 application	205
环境变量	208
application 的类型与终止策略	210
分布式 application.....	211
分阶段启动	215
内含型 application.....	217
内含型 application 的分阶段启动	217
将监督者与 application 组合到一起.....	219
SASL application.....	220
进度报告	224
错误报告	225
崩溃报告	226
监督者报告.....	227
总结	228
接下来是什么	229
 第 10 章 基于特殊进程打造自己的 behavior	 230
特殊进程.....	230
互斥体	231
启动特殊进程.....	232
互斥体的状态.....	235
处理退出	236
系统消息	237
跟踪与日志事件	238
合在一起	239
动态模块和休眠	243
属于你自己的 behavior	244

创建 behavior 时的要求	245
一个处理 TCP 流的例子	245
总结	249
接下来是什么	250
第 11 章 系统原则与发行包制作	251
系统原则	252
发行包目录结构	253
发行包资源文件	257
创建发行包	260
创建 boot 文件	262
打包发行包	271
启动脚本以及目标上的配置	275
参数和标志	277
init 模块	289
rebar3	290
生成一个 rebar3 发行包项目	292
使用 rebar3 创建发行包	295
使用 rebar3 处理制作发行包时的项目依赖问题	298
总结	300
接下来是什么	304
第 12 章 发行包升级	305
软件升级	305
第一个版本的咖啡机 FSM	308
添加一个新状态	311
为发行包创建升级	314
负责升级的代码	318
应用程序升级文件	322
高级指令	325
发行包升级文件	328
低级指令	330
安装升级	332
发行包处理器	334

升级环境变量.....	338
升级特殊进程.....	338
在分布式环境下升级.....	339
升级模拟器和核心 application.....	340
使用 rebar3 进行升级.....	341
总结.....	344
接下来是什么.....	346
第 13 章 分布式架构.....	347
节点类型与家族.....	348
联网.....	351
分布式 Erlang.....	353
套接字与 SSL.....	359
面向服务和微服务的架构.....	361
点对点.....	362
接口.....	364
总结.....	366
接下来是什么.....	367
第 14 章 永不停止的系统.....	368
可用性.....	368
容错.....	369
弹性.....	370
可靠性.....	371
数据共享.....	375
一致性和可用性之间的权衡.....	383
总结.....	384
接下来是什么.....	385
第 15 章 水平规模伸缩.....	386
水平规模伸缩与垂直规模伸缩.....	386
容量规划.....	390
容量测试.....	392
平衡你的系统.....	394

找寻瓶颈	396
系统蓝图	398
负载调节与背压	399
总结	401
接下来是什么	403
第 16 章 监视与抢救性支持	404
监视	405
日志	406
指标	411
警报	414
抢救性支持	416
总结	418
接下来是什么	420
索引	421

序言

本书为你提供的，是一名自 1996 年从 R1 版便开始接触 Erlang 的爱好者，钻研十多年后终于成长为一位分布式系统专家，在这一过程中他所获得的宝贵知识和经验，让你明白为何 Erlang/OTP 能够使你更容易地专注应对系统开发中那些真正的挑战。

通过描述如何构建 OTP 行为模式 (behavior) 以及为什么需要 OTP 行为模式，我们向你展示了如何使用它们构建独立节点。这就是最初我们向 O'Reilly 提供的草案，内容仅限于此。但是在编写本书时，我们决定将内容更进一步，记录下我们的实践经验、设计决策过程和架构分布式系统时常见的一些问题。通过我们所做的这一系列设计选择和折中，这些模式为我们提供了 Erlang/OTP 众所周知的可伸缩性、可靠性和可用性。与流行的观点相反，这一切并非魔法般地开箱即得的，但获得它们确实比其他任何——非语义级别模拟 Erlang 的，或者不是运行在 BEAM 虚拟机上的——编程语言要容易得多。

Francesco：为什么写这本书

有人曾告诉我，写书有点像生孩子。一旦你写完一本书，拿到纸质图书的那一刻，脑子里有的只是兴奋和激动，而曾经付出的艰辛将统统被忘掉，只渴望着赶快开始写另一本。自从 2009 年 6 月首次拿到纸质书以来，我一直有编写 *Erlang Programming* (O'Reilly) 续作的打算。在我开始这个项目时，我还没有自己的孩子，但最终这一项目花了如此长的时间以至于我的第二个孩子都已经快出生了。美好的事物值得我们等待，谁说不是呢？

与第一本书一样，本书是围绕我在 Erlang Solutions 公司所做的 OTP 培训材料中的示例编写的，我将使用这些示例时我的讲解和教学过程转化为文字。每当完成一章后，我都会回顾并确保那些学生较难理解的部分我的讲解是清晰的。最好的学生通常会问的那些问题最终被放到了补充材料部分，而篇幅较长的章则被分解成一些较短小的章。原本一切都很顺利，直到我们到达第 11 章和第 12 章时，因为发行包制作和软件升级没有一种统一的方法，而是存在许多种工具。有些工具需要集成到客户的构建和发布过程中，而其他一些则是开箱即用的，还有一些已无法使用。对于任何想要理解系统的发行包制作

和软件升级包括其幕后工作原理的人，我们希望这两章成为他们的终极指南。此外，如果你必须对现有工具进行故障排除或编写自己的工具，其中还介绍了你所需要了解的内容。

但真正的麻烦从第 13 章才开始。由于没有任何示例和培训材料，我发现自己必须将头脑中的内容形式化，将构建 Erlang/OTP 系统时所采取的方法落实为文档，并尝试将其与分布式计算理论结合起来。最终第 13 章变成了 4 章，并且花了写出本书前 10 章那么长的时间才完成。对于那些购买了早期访问（early access）的读者，我希望没有辜负你们的等待。对于那些明智地等我们写完才购买的朋友，希望你们喜欢这些内容！

Steve：为什么写这本书

我第一次发现 Erlang/OTP 是在 2006 年，当时我正在研究如何能更快、更便宜、更好地开发企业集成软件。无论我从哪个方面考察，Erlang/OTP 都明显优于我和我的同事当时一直使用的 C++ 和 Java 语言。2007 年，我加入了一家新公司，开始在商业产品中使用 Erlang/OTP，事实证明，我之前考察所发现的一切优势都是真的。我教一些同事使用了这种语言，不久后，我们开发的软件比其他大多数人开发的都更强大、更可靠、更容易演进，并且能更快地投入生产环境，甚至与人员规模大得多的 C++ 团队相比优势依然明显。直到今天，我仍然完全信赖 Erlang/OTP 在实践中表现出的令人印象深刻的高效性。

多年来我发表了不少技术资料，而我的目标读者一直都是像我这样的其他从业者。这本书也不例外。在前面的 12 章中，我们提供了许多深入的实践性细节，使开发人员能够充分理解 OTP 的基本设计原则。在这些细节中包含了大量极具实用价值的知识——各类模块、函数和方案等——它们将为你的日常设计、开发和调试工作节省大量时间和精力。在最后的 4 章中，我们将转变方向，聚焦于宏观的层面，探讨可伸缩分布式应用在开发、部署和运行时涉及的各种权衡取舍。由于与分布式系统、容错和 DevOps 相关的知识、方案和需考虑的权衡数量着实庞大，所以要想将这些章节简洁地写出来难度可想而知，但我相信本书为此达到了适当的平衡，既为读者提供了大量好的建议，同时又能避免读者迷失于其中。

我希望这本书能帮助读者提高所开发的软件和质量及效用。

本书的读者对象

本书的目标受众主要针对 Erlang、Elixir 开发人员和架构师——已经完整阅读过一本以上入门书籍，并准备将知识提升到一个新的水平的人。这不是一本带你入门的初等水平的书籍，而是一本涵盖了许多其他书籍未涉及的高级内容、让你远超同行水平的书。其中第 3 章至第 12 章存在依赖关系，应该顺序阅读，第 13 章至第 16 章也是如此。如果你不需要回顾 Erlang 初级知识，可以跳过第 2 章。

如何阅读本书

本书中的内容兼容 Erlang 18.2。书中涉及的绝大部分功能同样适用于先前版本的 Erlang；针对不适用的功能在书中均有指出。对于未来版本的不兼容性虽然在写书时尚不可知，但将会在本书的勘误页面上进行详细说明，并修复本书 GitHub 仓库中的对应代码。我们鼓励你从我们的 GitHub 仓库下载本书的示例代码，并亲自运行以更好地理解相关知识。

致谢

撰写本书是一个漫长的旅程。在进行这项工作时，我们得到了许多优秀人士的帮助。编辑 Andy Oram 给予我们无数的想法和建议，耐心地指导我们，给予我们反馈，并且不断鼓励我们。Andy，谢谢你，没有你，我们无法完成此书！Simon Thompson——*Erlang Programming* 一书的合著者帮助了本书的构思和起草，并为第 2 章奠定了基础。非常感谢 Robert Virding 贡献的一些例子。我们还得到了很多读者、审稿人、贡献者的帮助，为我们提供了许多反馈，有了这些，我们才能让每一章的内容充实。在此我们列出他们的姓名，并忐忑地希望没有遗漏任何一位：Richard Ben Aleya、Roberto Aloï、Jesper Louis Andersen、Bob Balance、Eva Bihari、Martin Bodocky、Natalia Chechina、Jean-François Cloutier、Richard Croucher、Viktória Fördös、Heinz Gies、Joacim Halén、Fred Hebert、Csaba Hoch、Torben Hoffmann、Bob Ippolito、Aman Kohli、Jan Willem Luiten、Jay Nelson、Robby Raschke、Andrzej Śliwa、David Smith、Sam Tavakoli、Premanand Thangamani、Jan Uhlig、John Warwick、David Welton、Ulf Wiger 和 Alexander Yong。如果我们在名单中遗漏了您，我们真诚地向您道歉！给我们发一封电子邮件，我们会将您添加进去。对 Erlang Solutions 职员中那些阅读了本书早期撰写过程中的手稿，以及其他早期为本书提供勘误的人，我们必须大声地向你们表示感谢。此外，还要特别感谢所有通过社交媒体渠道鼓励过我们的人，特别是其他作者。你知道我说的就是你们！最后，同样重要的一点是，感谢 O'Reilly 的制作、营销和会议团队，不断提醒我们“只要尚未付印，工作就不算结束”。我们非常感谢你们的支持！

本书所用的排版约定

本书中使用了以下排版约定：

斜体 (*Italic*)

用于表明新术语、应用程序、URL、电子邮件地址、文件名、目录名和文件扩展名。

等宽字体 (Constant width)

用于程序清单，以及在段落中对变量、函数名、数据库、数据类型、环境变量、语

句和关键字等程序元素的引用。还用于行为模式 (behavior)、命令和命令行选项等。

等宽加粗字体 (**Constant width bold**)

用于显示命令或用户手工输入的文本。

等宽斜体 (*Constant width italic*)

用于显示应该由用户提供或者根据上下文确定的值。



该图标表示提示或建议。



该图标表示一般性的备注说明。



该图标表示提醒或警告。

中文版书中切口以“◁□”表示原书页码，便于读者与英文原版图书对照阅读，本书的索引中所列的页码也为英文原版图书中的页码。

使用示例代码

补充材料 (示例代码、练习材料等) 可以从此网址下载：<https://github.com/francescoc/scalabilitywitherlangotp>。

本书的初衷是帮助你完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码。除非涉及大量引用代码，否则你无须联系我们获得许可。例如，编写程序时使用到了本书中的几个代码块，这不需要取得我们的许可。但其他情况例如销售或分发来自 O'Reilly 书籍中的示例代码则需要先取得许可。为了回答问题而引用本书中的内容和示例代码不需要许可。将本书中的大量示例代码整合到产品文档中则需要获得许可。

我们感谢但不要求注明出处。出处通常包括标题、作者、出版商和 ISBN。例如：“*Designing for Scalability with Erlang/OTP* by Francesco Cesarini and Steve Vinoski (O'Reilly). Copyright 2016 Francesco Cesarini and Stephen Vinoski, 978-1-449-32073-7.”

如果你觉得你对示例代码的使用不属于上述一般性合理情形，或者对许可范围存疑，欢

迎随时通过 permissions@oreilly.com 与我们联系。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

<http://bit.ly/designing-for-scalability-with-erlangotp>

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 <http://www.oreilly.com>。

在 Facebook 上找到我们：<http://facebook.com/oreilly>

在 Twitter 上关注我们：<http://twitter.com/oreillymedia>

在 YouTube 上观看我们：<http://www.youtube.com/oreillymedia>

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：你对书中内容的修改意见可在[提交勘误处](#)提交，若被采纳，将获赠博文视点社区积分（在购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方[读者评论处](#)留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/33747>



概述

你渴望实现一套容错、可伸缩、软实时的高可用性系统。它必须属于事件驱动型的——能够感知并应对外部状况（例如负载、故障等）的变化。你还期待它稳定可靠，不管何种状况都必须响应请求。而你已有所耳闻，Erlang 正是达成这一切的正确工具——许多案例确实已经证明了这一点。是的，Erlang 确实是一门非常强大的编程语言，然而单靠 Erlang 语言本身其实还不足以打造出你设想的这样一套极复杂的应变式系统。要想正确、快速、有效地完成这些工作，除了 Erlang 语言本身外，你还需要一些中间件、可重用的库、工具、设计原则，以及一套相应的编程模型——指导你如何架构和分布化你的系统。

本书的目标是基于 Erlang 语言与 OTP 框架为核心探索可用性和可伸缩性相关的主题，包括并发、分布式容错等。Erlang/OTP 诞生自爱立信计算机科学实验室，他们当时研究的目标是在上市周期紧迫的电信行业里如何做到高效地开发下一代电信系统。那个年代，许多如今我们熟悉的东西，比如 Web、平板电脑、智能手机、大规模多用户在线游戏、即时通信、物联网等都还不存在。

在那时，对可伸缩性和容错性具有如此严苛要求的系统只有电话交换机系统。因为这类系统必须处理新年前夜的峰值流量，监督并确保紧急呼叫服务正常，避免设备故障导致通信中断——那将使基础设施提供商面临可怕的高额违约罚款。换句话说，如果你拿起电话却听不到拨号音，你可以确定两件事，一是顶级管理层要有大麻烦了，二是这次通信中断必将成为报纸上头版头条的新闻。无论如何，这些电话交换机都不允许出问题。哪怕是周边部件和基础设施故障，通信服务请求也必须得到受理。时过境迁，过去害怕的是监管方的罚款，如今担心的则是没耐心的消费者——他们忠诚度不高，眨眼就能换掉供应商；而报纸上头版头条的谴责报道如今也变成社交媒体上铺天盖地的、歇斯底里的抱怨。是的，许多事已不同往日，但那些围绕可用性和可伸缩性的核心问题却一点也没变。

为了能够达到如此严苛的可伸缩性和容错性目标，电信交换机及类似的现代系统不仅

需要处理大量的负载变化和内部事件，还需要应对许多复杂的故障状况。最终爱立信计算机科学实验室的成员发现，为了解决这些问题，唯一的办法就是创建一门新的编程语言——而原本他们并没有这个打算。如今这已算得上一个经典例子，向大众展示了发明编程语言及编程模型对解决一个特定的、定义清晰的问题能起到何等显著的有益的作用。

定义问题

正如我们在本书中展示的那样，Erlang/OTP 在众多编程语言和框架中是如此独一无二，其广度、深度及其针对可伸缩的、容错的高可用系统提供的各种功能，以及在维持一致性方面都独树一帜。要知道，设计、实现、操作和维护这样的（可伸缩的、容错的高可用）系统难度很大。成功的团队是怎么做的呢？不断迭代上述四个步骤（设计、实现、操作、维护），通过持续度量产品性能和监视产品运行状况从而获得反馈，然后进一步找出可改进的方面——不仅是代码，更涉及开发、运维等各方面。成功的团队还学习通过其他手段来改进可伸缩性，例如测试、实验、基准度量等，他们还会跟进与他们系统各方面有关的研究与进展等。除此之外，决定团队是否能达成——甚至超出——既定目标的关键，也包含非技术问题如团队文化和价值观等方面。

我们使用了一系列术语，如“分布式”“容错”“可伸缩”“软实时”“高可用”等来描述我们打算使用 OTP 打造的系统。这些到底是什么意思呢？

“可伸缩”指的是一个计算系统可以适应负载和可用资源的变化。可伸缩的站点能做到，比如，平滑地处理峰值流量而不丢失任何客户请求，哪怕硬件故障也是如此。可伸缩的聊天系统可以适应每天数以千计的新增用户，同时不影响对已有用户的服务。

“分布式”指的是系统如何组成集群并相互交互的方式。集群既能通过添加硬件实现水平伸缩，也能在单台机器上，通过布置额外节点实例来更好地充分利用多个处理器的核心效能。这里说的“单台机器”有可能是虚拟化的，操作系统层叠运行或者相互并列共享裸机资源。请注意，通过添加更多的处理能力来扩充数据库集群可存储的数据量与每秒可处理的请求数很重要。但另一方面，收缩也同样重要，例如，一个构筑在云服务上的 Web 应用可能希望在峰值流量时扩容，之后当使用量下降时则尽快释放多余的计算实例。

“容错”的系统可以在其环境存在故障的情况下仍以可预期的方式运转。在一个系统设计的初期就必须把容错性考虑在内，别幻想过后再添加。如果你的代码中有错误或程序的状态出现错乱怎么办？或者如果你遇到网络中断或硬件故障怎么办？如果某用户发送

了一条消息导致一个进程崩溃，该用户将收到一条提示指明消息是否抵达目标，并且我们可以确保这一提示准确。

“软实时”指的是响应和延迟的可预测性，并且能维持恒定的吞吐量，保证响应在可接受的时间内做出。不管峰值流量多大、并发请求数多高，吞吐量都必须维持恒定。也就是说，无论同一时刻有多少请求正流经系统，吞吐量都不能因为负载太重而下降。响应时间——也叫延迟，应当做到只受并发请求数影响（只与其相关，而不受其他因素影响），避免处理不同请求时延迟出现较大的波动，例如因为垃圾回收器的世界暂停（stop the world）效应而受影响。如果你的系统的吞吐量是每秒 100 万条消息，而此时正有 100 万个请求等待处理，那么进行处理并将该请求递交给接收方只应耗时 1 秒。但如果处于峰值流量，有 200 万条请求发来，吞吐量也不能恶化，全部（绝不是部分）请求都必须在 2 秒内处理完。

“高可用”指的是把由于逻辑错误、通信中断、升级或其他运维活动所带来的停机时间降至最低，甚至完全消除。你需要考虑进程崩溃怎么办，数据中心的电力供应被切断了怎么办，你所准备的备用电源是否能支撑你完成集群迁移，网络资源和硬件资源是否也备好了足够的冗余量，你是否对系统做了规划——确保即使集群中一部分硬件失效，剩余的硬件也拥有足够的 CPU 容量应对峰值负载。无论何种情形——基础设施出现局部故障、云服务器提供商内部出现尴尬的通信中断，或你正对系统做维护——系统的用户就是这样永远期待一切正常。这与“容错”不同，容错只须做到告知用户任务失败，同时系统本身不受影响继续运行。Erlang 拥有边运行边升级的能力，有助于你从“容错”进阶到“高可用”。不过事情也不是这么简单，特别是考虑到升级过程可能涉及数据库模式变化，或者是在分布式环境下进行协议升级，而新协议不能向下兼容老协议，同时又要求升级过程中还需要继续处理请求的时候，难度就层层累加了。总而言之，高可用就是当你周末或者晚上使用网上银行的时候，不会看到银行网站上贴出“因例行维护暂时关闭”之类令人尴尬的提示。

Erlang 确实有助于解决此类问题。不过归根结底，它毕竟只是一门编程语言。对于你想实现的复杂系统，你需要有一些开箱可用的预构建好的应用程序和库。你还需要一些架构方面的设计原则和设计模式辅助你创建分布式的、可靠的集群。你需要一些指南，指明该如何设计你的系统，同时提供一些工具来实现、部署、监视、操作和维护它。本书中我们涵盖了一些库和工具，它们允许你把错误隔离在节点级别，并创建和分布多个节点，以实现可伸缩性和可用性。

你需要认真考虑你的需求及你拥有的资源，确保选择了正确的库和设计模式，这样才能

确保最终的系统是按你最初设想的那样去运作。在不断探索的过程中，你将不得不面临一些取舍，它们相互制约——时间、资源、功能，甚至可用性、可伸缩性、可靠性等。如果你自己都还没搞清楚到底要对系统进行何种取舍，那就不要奢望有什么现成的库可以帮助你。在本书中，我们通过一系列步骤来引导你理解这些需求，然后再通过一些步骤让你明白为了实现这些需求需要做出怎样的设计抉择以及取舍。

OTP

OTP 由一组独立于特定领域的框架集、设计模式构成，它引导并支持你设计、实现、部署基于 Erlang 的系统。在项目中使用 OTP 可以帮你避免意料之外的复杂性——那些由于工具选择不当而引发的困难。除此之外，其余问题的难度就与工具无关了——无论选用什么编程工具或中间件。

爱立信很早便意识到这一点。在 1993 年，在开发第一款 Erlang 产品的同时，爱立信启动了一个项目，这个项目用于解决工具、中间件、设计原则方面的难题。开发人员希望避免那些意料之外的复杂性，特别是这些复杂性已经被解决过了，这样他们可以把更多的精力集中在解决真正困难的问题上。于是 BOS——Basic Operating System 诞生了。1995 年，BOS 项目与 Erlang 项目合并了，二者合二为一构成了今天我们熟知的 Erlang/OTP。也许你曾经听说过 Erlang 背后的支持团队叫作 OTP 团队。这一“梦之队”正是此次合并的成果，Erlang 从研究团队移出来的同时，爱立信组建了一个产品团队来进一步开发和维护它。

传播 OTP 相关的知识可以让更多信奉“实践检验真理”（*tried and true*）原则的公司的 IT 部门采用 Erlang。让大家知道存在一个稳定且成熟的平台可用于应用程序开发，且有助于技术人员说服管理层接受 Erlang，这是让更多行业人员接纳的关键一步。初创公司更乐于接受 Erlang，正因为 Erlang/OTP 可帮助他们更快地将产品投入市场，并减少开发和运维成本。

OTP 由三大部分构成（参见图 1-1），将这三大部分协同运用，可为我们解决前文提及的难题，为设计和实现相应的系统提供可靠的方法。这三大部分分别是 Erlang 语言本身，一组相关工具和库，以及一组设计原则。接下来我们逐个进行介绍。

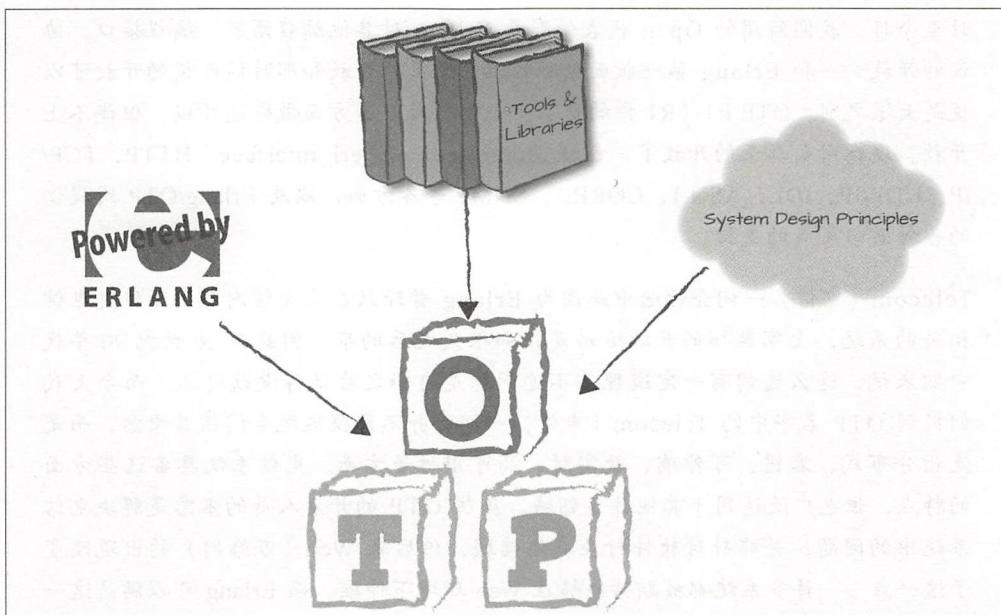


图1-1: OTP组件

为什么取这个名字

OTP 代表什么意思？与其我直接告诉你，不如你先猜一猜。假如你尝试以 OTP 为关键词进行搜索，你可能会看到 OTP 指的是 One True Pair（完美情侣）。如果你是一个想象力丰富的人，没准会猜 Oh This is Perfect（无懈可击）、On The Phone（电话之上）、Open Transaction Platform（开放事务平台）等。还有些人可能会认为 OTP 是 Online Transaction Platform（在线事务平台）的缩写——但那一般习惯缩写成 OLTP，而不是 OTP，因此也不对。最离谱的猜测是说，之所以叫 OTP 只是为了让 Erlang the Movie 系列显得更酷。哎呀，这些都不对。OTP 其实是 Open Telecom Platform（开放电信平台）的缩写，这个名称是由 Bjarne Däcker 创造的，他是爱立信计算机科学实验室的前领导人，Erlang 正是在那里诞生的。

Open（开放）在 20 世纪 90 年代中期的爱立信公司里是一个流行词。不管是什么都得 Open，比如 Open System（开放系统）、Open Hardware（开放硬件）、Open Platform（开放平台）等。爱立信的市场部门甚至打印了一张开阔的风景海报把它挂在走廊里，标题就叫 Open System。然而实际上没人真的知道此处的 Open System 以及其他各种 Open 到底是什么意思，但这个词在当时很流行，所以与其戳穿还不如趁此机会也当一次时尚人士，于是 OTP 里的 Open 就这么无脑地被加进去了。

时至今日，我们所谓的 Open 代表的则是 Erlang 对其他编程语言、编程接口、协议的开放——和 Erlang 第一版面世时相比，今天的开放和那时候所谓的开放可以说是天壤之别。OTP R1（R1 指的是第一个发行版）各方面做得还可以，但谈不上开放。现在则是真正的开放了，包括 JInterface、ei、erl_interface、HTTP、TCP/IP、UDP/IP、IDL、ASN.1、CORBA、SNMP 等各方面，以及 Erlang/OTP 所提供的各种面向集成的支持。

Telecom（电信）一词会被选中是因为 Erlang 曾经只在爱立信内部用于研发电信相关的系统，大家熟知的开源运动是在那很久之后的事。因此在 20 世纪 90 年代中期来说，这么选词有一定道理，不过问题是自那之后就再没改过名，而今天我们提到 OTP 名字中的 Telecom（电信）一词时并不再狭隘地专门指其原意，而是泛指分布式、容错、可伸缩、软实时、高可用性等方面。电信系统具备这些方面的特点，但也广泛适用于其他许多领域。虽然 OTP 的开发人员的本意是解决电信系统中的问题，并非针对软件行业其他领域，但后来 Web（万维网）的出现改变了这一点——许多系统都被期待能够在 Web 环境下伸缩。而 Erlang 可以满足这一点——甚至在 Web 还没发明前就已经可以满足这一点！

OTP 中的最后一个词 Platform（平台），尽管有些平淡直白，却是唯一一个真实描述了 OTP 的词。选这个词的时候，爱立信管理层正打算开发各种平台。所有软件相关的东西都必须基于（最好是开放的）平台进行开发。

所以实际情况就是，Bjarne 选了一个有一定含义的词，这个词可以取悦高层，确保他们会持续资助这一项目。其实高层可能根本不清楚这个计算机科学实验室是做什么的，会带来什么样的麻烦，不过最起码他们愿意容忍。

自从 1998 年 Erlang/OTP 以开源方式发行后，很多关于改名的讨论出现了，但最后也没结论。在早些时候，电信部门之外的开发人员错误地跳过了 OTP 而只使用 Erlang，因为——按他们的原话——他们“不是在开发电信应用”。社区和爱立信如今已决定会一直沿用 OTP 这一名称，弱化电信一词，但依然强调其重要性。这样的构成看起来算得上合理。

Erlang

第一大部分是 Erlang 自身，包括语言的语义及下层的虚拟机。关键语言特性，如轻量级进程、缺少共享内存及异步消息传送，会让你向目标更进一步。同样重要的还有进程间的链接和监视，以及用于错误信号传播的专门通道。监视加上错误报告允许你相对容易地构建出复杂的监督层级，同时带有内置的错误恢复。因为消息传送和错误传播都是异步的，基于其开发的系统的语义和逻辑可以轻松地从运行在单一节点上改为分布式方式，

而且无须修改任何代码。

在单节点上运行与在分布式环境中运行有一点显著不同，那就是投递消息和错误时的延迟。但在一个软实时系统内，你必须考虑延迟问题，不管系统是否是分布式的或者是否处于高负载下。所以如果解决了这两类问题的其中一类，另一类问题同时也就解决了。

Erlang 把你的代码放在一个针对并发高度优化过的虚拟机上运行，其针对每个进程单独执行垃圾回收，优势是可预测性及简单的系统行为。其他编程环境就享受不到这一点了，因为它们不得不用额外的一层来模拟 Erlang 的并发模型和错误语义。引用 Joe Armstrong (Erlang 联合发明人) 的一句话：“你可以模拟 Erlang 的逻辑，但只要不是真正运行在 Erlang 虚拟机上，你就无法模拟其语义。”时至今日，能绕过这一障碍的几门编程语言都是基于 BEAM 模拟器——一种当下主流的 Erlang 虚拟机——构建的。这几门语言都有自己的生态系统，直至本书编写时，Elixir 和 Lisp Flavored Erlang 人气最旺。

工具和库

第二大组成部分包括了一些 application (应用)，早在开源成为软件项目常态之前就随标准 Erlang/OTP 套件一起分发了。你可以把每个应用视为 OTP 打包资源的一种方式，应用间可以存在依赖关系。具体来说，有包括工具、库、面向其他语言的接口、编程环境、数据库、数据库驱动、标准组件及协议栈等各方面的应用。OTP 文档很好地把它们归纳为如下几类。

- 基础应用包括如下几项。

- Erlang 运行时系统 (*erts*)
- 核心 (*kernel*)
- 标准库 (*stdlib*)
- 系统架构支持库 (*sasl*)

当需要创建、启动、升级系统时，这些应用将作为重要部件或辅助工具为你提供支持。在本书中，我们会介绍应用的具体细节。这些应用加上编译器本身，是你在开发基于 Erlang/OTP 的系统时必定会用到的部分，因此很重要。

- 数据库方面的应用有 *mnesia* —— Erlang 的分布式数据库；以及 *odbc* ——用于与关系型 SQL 数据库通信。*mnesia* 受到许多人的喜欢，因为它速度快，与你的应用程序在相同的内存空间内运行和存储数据，上手简单，通过 Erlang API 即可访问。
- 运维方面的应用有 *os_mon*，它使你能够监视底层的操作系统；*snmp*，SNMP 协

◀ 8

议 (Simple Network Management Protocol) 代理与客户端; *otp_mibs*, 一款管理信息库, 可以用 SNMP 协议来管理你的 Erlang 系统。

- 接口集合以及通信应用提供了协议栈和接口, 可用于与其他编程语言协同工作。其中包括一个 ASN.1 (*asn1*) 编译器及相应的运行时支持, 可直接接入 C (基于 *ei* 和 *erl-interface*) 和 Java (基于 *jinterface*) 程序, 还带了一个 XML 解析器 (*xmerl*)。另外还有一些安全应用, 主要涉及 SSL/TLS、SSH、密码学、公钥基础设施等方面。图形包里包括了一个移植版的 *wxWidgets* (*wx*), 及对应的易用的接口。*eldap* 应用则针对轻量级目录访问协议 (LDAP) 提供了一套客户端接口。并且对电信爱好者来说, 有一个 *Diameter* 栈 (遵循 RFC 6733), 用于策略控制和授权, 及认证与计费。再深挖一些, 你会找到 *Megaco* 栈。*Megaco/H.248* 是用于控制物理分解多媒体网关的元素的协议, 将媒体转换与呼叫控制分开。如果曾经用过智能手机, 你很可能已经间接地接触过 Erlang 的 *diameter* 和 *megaco* 这两个应用了。
- 下列工具应用能加速你的 Erlang 系统开发、部署和管理。本书中只涉及了一部分, 在这里把它们概述一下:
 - *debugger* 是一个图形化的工具, 允许你追踪代码执行步骤, 了解其是如何影响函数状态的。
 - *observer* 整合了应用程序监视器、进程管理器及其他基础工具, 让你可以在开发过程中或产品环境下监视你的 Erlang 系统。
 - *dialyzer* 是一个静态分析工具, 可找出类型矛盾、死代码及其他问题。
 - 事件追踪器 (*et*) 使用端口来收集分布式环境下的事件, *percept* 则允许你通过追踪并可视化并发相关的活动来定位瓶颈。
 - Erlang 语法工具 (*syntax_tools*) 包含一些模块用于处理 Erlang 语法树, 其处理方式兼容其他语言相关工具。它还包括了一个模块合并器, 允许你合并 Erlang 模块, 同时还带了一个重命名器, 用于解决非层次化模块空间带来的冲突问题。
 - *parsetools* 应用包括针对 Erlang 的解析生成器 (*yecc*) 和词法分析生成器 (*lecx*)。
 - *reltool* 是一个发行 (release) 管理工具, 其提供了图形化的前端, 并在后端开放了一些钩子 (hook) 可供更通用的构建系统 (build system) 使用。
 - *runtime_tools* 是一个工具集, 其中包含 DTrace 和 SystemTap 探针, 以及 *dbg* —— 一个将内置函数追踪功能包装得更易用的工具。

— 最后, *tools* 应用是由剖析器 (profiler)、代码覆盖率分析工具及模块交叉引用分析工具构成的集合, 其中还附带了用于扩展 emacs 编辑器的 Erlang 模块。

- *test* 应用程序提供了单元测试 (*eunit*)、系统测试和黑盒测试方面的工具。其中的 Test Server (打包在 *test_server* 应用里) 作为框架可以在更高层的测试工具应用程序里充当引擎。其实你没必要直接用它, 因为 OTP 提供了更高层的测试工具 *common_test*——一个适合用于黑盒测试的应用。*common_test* 支持针对大多数目标系统——无论其编程语言是什么——自动执行基于 Erlang 的测试用例。
- 为了怀念已逝的时光, 在此我们还想特别提到两个应用, 一个是 *orber*, 它用于 Object Request Brokers (ORB) 方面; 另一个是 *ic*, 它是一个 Interface Definition Language (IDL) 编译器。另外还有一些已经不再使用的与 CORBA Common Object Services 相关的应用, 这里就略过了。

上述应用程序中的一些我们在本书中会涉及。另有一些本书没讲的在 *Erlang Programming* 一书中有所涉及, 剩下的则可以在标准 Erlang/OTP 文档中的参考手册页及用户指南中找到。

社区实现了数以千计的开源应用程序用于增强 Erlang。我们在本书中的后半部分会涉及其中较流行的一些应用程序, 它们聚焦于分布式架构、可用性、可伸缩性以及监视。如 Riak Core (https://github.com/basho/riak_core) 和 Scalable Distributed Erlang (<http://www.dcs.gla.ac.uk/research/sd-erlang/>) 框架; 负载规则应用程序 *jobs* 和 *safetyvalve* 以及监视及日志应用程序 *elarm*、*folsom*、*exometer* 和 *lager* 等。一旦你读完此书, 在开始你的项目前, 看看标准的开源 Erlang/OTP 参考手册和用户指南吧, 因为你永远不知道它们何时会派上用场。

系统设计原则

10

OTP 的第三大部分则由一系列 principle (抽象原则)、rule (设计规则) 和 behavior (行为模式) 构成。抽象原则描述的是 Erlang 系统的软件架构, 其基本元素是进程, 更进一步说是基于通用行为模式的进程。设计规则的作用是保证兼容性——你应当使用与开发系统相兼容的工具。通过使用这种方法为你解决问题提供了一种标准化的方式, 让你的代码更容易理解与维护, 同时还在团队中建立了共通的语言与词汇表。

可以把 OTP 的通用行为模式 (generic behavior) 看作一种形式化的并发设计模式, 这些行为模式被打包放在库模块里, 其中含有一些通用的代码, 其可以解决一些有共性的问题。这些行为模式内置了对调试、软件升级、通用错误处理等各方面的支持。

遵循通用行为模式的进程既可以是 worker (工作者) 进程——完成主要复杂任务的那类

进程，也可以是 supervisor（监督者）进程——只负责启动、停止或监视其他 worker 或 supervisor。因为 supervisor 可以监督其他 supervisor，应用内的功能之间可以相互衔接，这样一来，开发工作可以轻松地以更加模块化的方式进行。被 supervisor 监视的进程称为 supervisor 的 children（孩子）。

OTP 针对 worker 和 supervisor 提供了预定义的库，允许你专注于系统的商业逻辑。我们把进程结构化为层次化的监督树（supervision tree），获得了容错的结构——隔离了错误，有助于恢复。OTP 允许你把监督树打包到应用程序内，如图 1-2 所示，其中双环圆圈代表 supervisor 进程，其余进程为 worker。

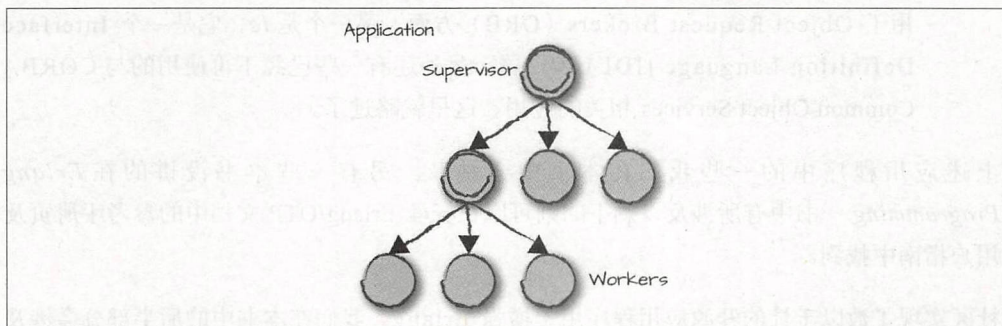


图 1-2: OTP应用

11 包含在 OTP 中间件中的通用行为模式包括如下几项。

- `gen_server`，通用服务器，提供了 C/S（Client/Server，客户端 / 服务端）设计模式。
- `gen_fsm`，通用有限状态机，允许你实现 FSM（Finite State Machine，有限状态机）。
- `gen_event`，通用事件处理和管理模式，允许你以通用的方式处理事件流。
- `supervisor`，监督者模式，监视其他 worker 或 supervisor 进程。
- `application`，应用模式，允许你把包括监督树在内的资源打包为一体。

我们在本书中会涉及上述全部内容，并教你如何自己实现它们。我们使用上述行为来创建 supervision tree（监督树），然后将其打包至应用内。然后把多个应用打包在一起构成一个 release（发行版）。每个 release 描述了 node（节点）内所运行的东西。

Erlang 节点

每个 Erlang 节点（node）内含若干个松散耦合的应用，其中可能有来自本章前面“工具和库”一节中所描述的应用，以及其他第三方应用，还有你为了实现系统而特别编写的应用等。这些应用之间既可能相互独立，也可能某些应用依赖于另一些应用的服务和 API。图 1-3 解释了一个典型的 Erlang 节点及其虚拟机，和该虚拟机依赖的硬件和操

作系统，以及运行在该虚拟机上的 Erlang 应用，还包含了非 Erlang 编写但与 Erlang 有交互的组件（它们与 OS 和硬件相关）。

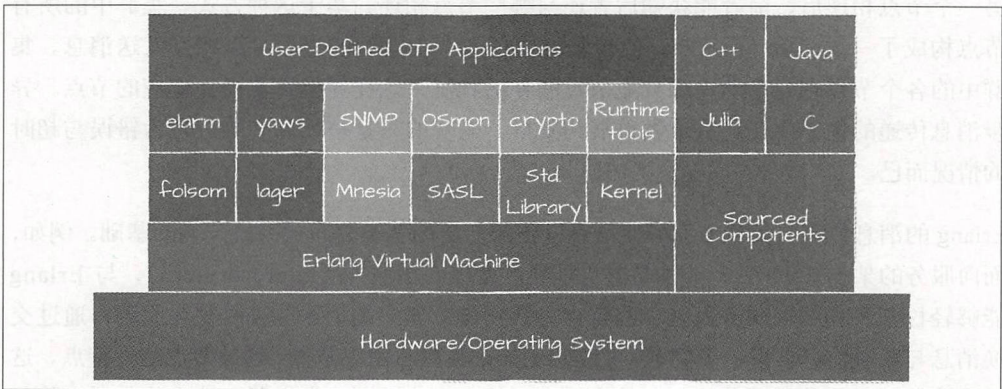


图1-3: 单个Erlang节点

把一群 Erlang 节点组合起来——其中部分节点是其他编程语言编写的也可以——就得到了一个分布式系统。现在你可以通过添加节点来扩展系统了，直到你遇到某些物理限制。具体就看你是怎么共享数据的，硬件或网络存在什么样的限制，或者外部依赖部分是否有瓶颈。

◀ 12

分布式、基础设施、多核

Erlang 源于电信行业，而这一行业与生俱来的需求就是容错，实现容错的关键就是分布式。如果没有分布式，一个单机上运行的应用其可靠性和可用性将严重依赖于构成该机器的硬件和软件。一旦该机器上的 CPU、内存、持久化存储、外围设备、电源供应或基板出现任何问题都能轻易地让整个机器挂掉，更别提上面运行的应用了。类似的，主机操作系统或支持库方面的问题也能让应用挂掉，或使其功能异常。分布式为我们调和这类矛盾指明了道路。

十多年来，计算机工业不断探索编程语言支持分布式的各种方法。设计一门通用的语言已经够难了，还要把它们设计成支持分布式，更进一步提高了难度。正因如此，常规套路是通过 library（库）的方式为非分布式的编程语言添加分布式支持。这种方式的好处是允许分布式的支持问题独立于语言而发展演化，但常常不得不忍受后加入的分布式支持与已设计好的语言间的阻抗不匹配问题，这让开发人员感到很为难。由于绝大多数语言使用函数调用作为在应用程序内各部分间传递控制与数据的主要手段，因此后期附加而上的分布式库也把分布式系统中各应用程序之间的数据交换建模为函数调用。

在 Erlang 里，进程间通过异步消息传递来通信。这种方式即使目标进程处于远程节点上也行得通，因为 Erlang 虚拟机支持从一个节点将消息传递到另一个节点。当一个节点与另一个节点相连后，前者能获知后者还与哪些节点相连。基于这种方式，集群中的所有节点构成了一个网格（mesh），使得集群内任意节点之间的进程可以相互发送消息。集群中的各个节点也能自动追踪其他节点的存活状况，这样便能发现失去响应的节点。异步消息传递的优势体现在从单机扩展为集群，改变的只是接收到的响应包含错误与超时的情况而已。

13

Erlang 的消息传递与集群方面的原语可以作为广泛的各类分布式系统架构的基础。例如，面向服务的架构（SOA），特别是其更现代的变体，微服务（microservices），与 Erlang 能够轻松开发和部署服务式进程的特点尤其吻合。客户端把这些进程视为服务，通过交换消息与其通信。再举一个例子，想想 Erlang 集群不需主节点/领导节点这一特点，这就意味着可以将其用于基于复制的点对点系统。客户端可以发送服务请求给集群内的任意节点，该节点要么自己完成处理，要么路由给其他节点。这种独立集群的概念，被称为组（group），组与组之间通过网关（gateway）节点相互通信，网关节点的数量可以动态增加或减少，并且相互之间的连接允许断开，有一个名为 *SD Erlang* 的框架把这一切都实现了。另一个流行的分布式框架，受 Amazon Dynamo 2007 年发表的论文（<http://bit.ly/riak-dynamo>）启发而来，名为 *Riak Core*，其提供了用于调度任务的一致性哈希算法，能通过一致性哈希算法从分区网络及失效节点上恢复，符合最终一致性，并提供了虚拟节点机制把状态和数据划分为小的、可管理的实体，从而可跨节点复制或移动。

借助分布式系统，你可以达到可伸缩性。事实上，可用性（availability）、一致性（consistency）和可伸缩性（scalability）三者环环相扣，相互影响。从单节点上的并发模型与消息传递概念为起点，我们又将其扩展，跨越网络，用于集群。Erlang 的虚拟机利用了今天多核系统的优势，允许进程以真正并发的方式运行在不同的核心上。由于 Erlang 虚拟机拥有“对称多处理器”（SMP）方面的能力，Erlang 已做好准备帮助应用程序随着每片 CPU 核心数的增加而垂直伸展。并且由于在集群上添加节点是如此轻松——新加入的节点只需访问一个已有的节点便能加入整个网络——水平伸缩问题也就迎刃而解了。终于，你可以聚焦于解决分布式系统中那些真正的挑战了，即，如何将你的数据和状态分布在不可靠的主机和网络上。

总结

为了让设计、实现、操作和维护更轻松可靠，你的编程语言和中间件必须简捷，它们运行过程中的行为必须可预测，并且最终编写出的代码应当可维护。我们一直在讨论容错、可伸缩、软实时，还要求高可用，这并不是说 Erlang/OTP 给你带来的好处只有当你面对

如此复杂的问题时才有所体现。即使你是在 Parallela board、BeagleBoard、Raspberry Pi 上做嵌入式开发，Erlang/OTP 的优势也是显而易见的。你会发现，Erlang/OTP 无论是对于嵌入式设备开发——作为其上的编制（orchestration）代码，还是对服务端开发——天生并发的环境，以及其他所有通向可伸缩、分布式多核架构与超级计算机道路的领域，都是很适合的。它让较难的软件问题变得容易，更让相对简单的程序变得更容易实现。

通过本书你将学到什么

14

全书分为两部分。第一部分，第 3 章至 10 章关注的是单节点上的设计和实现。建议按顺序阅读这几章，因为前后章节中的例子和解释相互有关联。第二部分，第 11 章至第 16 章，聚焦于与部署、监控、运维相关的工具、技术和架构，其中还阐述了一些技术方案用于解决可靠性、可伸缩性和高可用性方面的问题。第二部分涉及的例子在第一部分中也有涉及，但这两部分内容是可以独立阅读的。

我们在第 2 章中以 Erlang 概况作为开始，目的并非是向你从头教授这门语言，而只是一个温习。如果你对 Erlang 还很陌生，建议你首先学习一两本写得好的基础书籍来熟悉这门语言。比如 Simon St. Laurent 所著的 *Introducing Erlang*，Francesco Cesarini 和 Simon Thompson 所著的 *Erlang Programming*，或者其他我们在第 2 章里提到的书。我们在第 2 章里撰写的 Erlang 概况涉及这门语言的主要核心元素——列表、函数、进程、消息、Erlang 交互式命令行控制台，以及其他一些使得 Erlang 在众多语言中显得独一无二的特性，如进程链接和监控、热升级和分布式等。

Erlang 概况介绍完后，第 3 章讨论进程结构。Erlang 进程可以处理各种各样的任务，每种任务都有自己独特的需要解决的问题，但不同任务之间所拥有的相似代码结构以及进程生命周期将渐渐浮出水面，类似于我们在流行的面向对象语言，如 Java 和 C++ 中所观察到并总结出的常见设计模式那样。OTP 把这些常见的以进程为核心的结构和生命周期总结提炼为 behavior（行为模式），这些行为模式是 OTP 框架中实现代码重用的基本元素。

在第 4 章，我们继续深入探索前面引入的第一个 worker 进程。它实际上是 OTP 里最常见也是最常用的一种行为模式，名为 `gen_server`。正如其名，它支持通用的客户端/服务端结构，这种结构一般来说是服务端管理着一些特定的计算资源——例如简单的 ETS（Erlang Term Storage）实例，或网络连接池（连接池中的连接是与非 Erlang 构建的其他服务器相连的）等——并授权客户端访问这些资源。客户端与通用服务器之间既可以通过 `call-response`（调用 - 响应）的方式同步通信，也可以通过 `cast`（投递）方式发送单向消息异步通信，还可以通过普通的 Erlang 消息原语进行通信。为了全面理解这些通信模式，我们必须仔细观察与进程相关的各个方面，例如消息交换过程中客户端/服务器崩溃时会发生什么，超时机制如何应用于其中，服务端收到无法理解的消息时会发生什么

等。针对此类以及其他常见问题, `gen_server` 在独立于特定问题域的情况下给出了解决方案, 使得开发人员可以把时间和精力更多地花费在他们的应用开发上。`gen_server` 这一行为模式用处很大, 除了用在各种 Erlang 应用程序里, 在 OTP 自身中也到处是它的身影。

在进一步介绍更多 OTP 行为模式前, 我们先借刚讨论过的 `gen_server` 去看看 OTP 行为模式所提供的一些控制和监察点 (参见第 5 章)。这些功能反映了 Erlang/OTP 与其他语言和框架不同的地方: 内置了监察能力。如果想要知道你的 `gen_server` 现在正在做什么, 可以从 Erlang Shell 下简单地开启针对该进程的调试追踪功能, 并且既可以在编译时开启也可以在运行时开启。启用追踪将使其生成信息指明其收到了什么消息以及采取了何种动作来处理这些消息。Erlang/OTP 还提供了用于剖析运行中的进程的功能, 可查看其堆栈状态 (backtrace)、进程字典、父进程、链接进程以及其他细节。还有一些 OTP 功能可用于检查状态以及针对行为模式和其他系统进程的内部状态。正因为有这些面向调试的特性, Erlang 程序员常常忘了使用传统的调试器, 而是靠追踪来帮助他们诊断故障程序, 这种方式更便捷而且信息更加丰富。

我们接下来考察另一个 OTP 行为模式: `gen_fsm` (参见第 6 章), 它支持通用的有限状态机 (FSM) 模式。你可能已经知道了, 有限状态机是一种只由有限数量的状态构成的系统, 进入这一系统的消息将使该系统从一种状态转移到另一种状态, 状态转移过程可带有副作用。例如, 你可以把你的电视机机顶盒视为一个有限状态机。其状态包括当前选择的频道是哪个、是否显示屏幕画面 (on-screen display, 这里指的应该是机顶盒系统的叠加选单画面) 等。通过按下遥控器上的按钮, 可以改变机顶盒的状态, 比如切换到另一个频道, 或者改变其屏幕画面以显示频道指南或按需付费列表等。有限状态机广泛用于各种问题域, 原因是开发人员借助它可以更轻松地确定应用程序存在哪些潜在的状态以及这些状态间有哪些转换, 进而能够实现它们或归因出现的故障。知道什么时候以及如何使用 `gen_fsm` 可帮你避免自制粗糙的状态机, 因为这种临场发挥的设计往往随着时间推移会变成难以维护和扩展的意大利面条式代码。

一个系统要成功实现可伸缩, 完善的日志和监控功能必不可少。它们能够收集系统运行过程中的重要信息, 指明哪些地方存在瓶颈和问题, 有助你进一步探查各方面的问题。Erlang/OTP 的 `gen_event` 行为模式 (参见第 7 章) 提供了消息流子系统方面的支持, 此类子系统产生并管理消息流, 当系统中出现可能产生冲击性状态改变时将向你反馈, 例如 CPU 负载持续飙高、队列堆积、集群内节点断开等。它能够处理应用内各式各样的事件, 包括来自用户交互、传感器网络、第三方应用的等。我们不但会介绍 `gen_event` 行为模式的各个方面, 还会涉及 OTP 的 SASL (System Architecture Support Library) 系统架构支持库中提供的错误日志事件处理器等, 这一功能很灵活, 可以管理 supervisor 产生的各种报告, 例如普通报告、崩溃报告、进度报告等。

注意，事件处理和错误处理是许多编程语言的基本功能，它们在 Erlang/OTP 里也极其有用，但 Erlang/OTP 处理错误的方式与大多数程序员习惯的方式有极大的区别，千万别搞混。

介绍完了 `gen_event`，我们研究的下一个行为模式是 `supervisor`（参见第 8 章），它是用来管理 `worker` 进程的。在 Erlang/OTP 里，`supervisor` 进程启动 `worker`，然后监督它们执行应用程序任务。一个或多个 `worker` 可能会意外死亡，`supervisor` 可以以某种或多种方式来应对这些情况，具体细节我们在后面再解释。这种错误处理方式，被称为任其崩溃（let it crash），和大多数程序员采用的防御式编程策略有显著区别。任其崩溃加上监察机制，共同构成了 Erlang/OTP 的基石，实践证明极其有效。

然后我们看看最后一个 OTP 基础行为模式——`application`（参见第 9 章），它是将 Erlang/OTP 运行时系统（runtime）和你的代码整合在一起的汇聚点。每个 OTP `application` 都有相应的配置文件，指明其名称、版本、模块、依赖哪些其他 OTP `application` 等细节。当被 Erlang/OTP runtime 启动后，你的 `application` 转而启动一个顶级 `supervisor` 再进一步启动程序的其余部分。将代码模块化为 `application` 有助于你实现代码热升级。通常一份打包好的 Erlang/OTP 发行包是由许多 `application` 构成的，其中一部分属于 Erlang/OTP 开源分发包自带的，另一部分则是你编写的。

讲解完了各个标准行为模式后，我们把注意力转向如何编写自己的行为模式和特殊进程方面（参见第 10 章）。特殊进程是一类遵循特定设计规则的进程，它们可以被添加到 OTP 监察树中。了解这些设计规则不仅有助于理解标准行为模式的实现细节，而且可让你明白这些行为模式背后的折中考量，这样你就清楚什么时候该用标准行为模式，什么时候该自己写。

第 11 章描述了单个节点上的多个 OTP 应用如何捆绑在一起作为一个整体启动。你得创建自己的 `release` 文件——Erlang 世界里我们称之为 `rel` 文件。然后借助 `systools` 模块将 `rel` 文件中列出的对应版本的应用程序和运行时系统捆绑在一起，附带上虚拟机后放在单独的 `release` 目录里。这个 `release` 目录再经过配置和打包，就可以运行在目标系统上了。我们将介绍社区贡献的 `rebar3` 和 `relx` 工具，它们是构建代码和制作发行包的最佳方式。

部署你的自制系统时要注意，Erlang 虚拟机支持配置系统资源限额等选项设定。这些选项很多，从 ETS 表上限、进程上限到代码搜索路径、加载模块时的模式等。Erlang 里的模块既可以在启动时加载，也可以在首次调用时加载。在带有严格版本控制的系统里，你必须把它们运行在嵌入式模式（`embedded mode`）下，在这种情况下启动时便会进行模块加载，如果找不到模块将报错崩溃；而在交互式模式（`interactive mode`）下，如果模块尚未加载，并不会立刻终止进程，而是先尝试进行加载。在 Erlang 虚拟机外部，有一

◀ 17

个心脏监控 (monitoring heart) 进程, 它通过发送心跳的方式对 Erlang 虚拟机进行监控, 一旦发送的心跳收不到回应, 便会调用一段由你设计的脚本。你通过亲自实现这段脚本, 来决定是简单地重启当前节点还是根据重启节点的历史情况把崩溃级别提升到终止虚拟机甚至重启整台机器的级别。

Erlang 的动态类型允许你在运行时升级你的模块, 并且整个过程保持进程状态不变。但它不清楚如何解决模块间依赖、进程状态更改、协议变得不再向下兼容等方面的问题。OTP 提供了工具帮助你在应用程序甚至是运行时层面实现系统级 (system level) 升级。相关的原则以及支持库在第 12 章中进行介绍, 从定义你自己的应用程序升级脚本到编写支持发行文件升级的脚本都有涉及。还提供了应对数据库模式更改的方法和策略, 以及在分布式环境下协议变得不再向下兼容时的升级指南。在分布式环境下修复错误、改进协议、变更数据库 schema、运行时升级这些原本是让人心惊胆战的事, 但这些工具是如此强大, 让我们可以自动化地、不停机地完成升级。你可能在过去曾经碰到过网上银行因为维护而不能访问的情况, 如今不应该再出现这种情况了, 否则请把本书发给银行的 IT 部门。

为了运行和维护系统, 我们需要能够看到正在发生的事情。为了对集群进行伸缩, 我们需要一些策略来确定如何共享数据和状态。为了实现容错, 需要一些方法来实现如何复制和持久化。完成这一切的同时, 我们还得应对不可靠的网络, 考虑出错时的恢复策略等。尽管这些主题已经需要单独写一本书来探讨了, 我们在本书的最后一章还是提供了一些理论背景, 当你想把你的系统变为分布式使其可靠和可伸缩时, 这些知识会有用。我们是以描述在 Erlang/OTP 中设计一个可伸缩的高可用的架构所需的步骤的方式来介绍这些理论的。

第 13 章将会概述在设计分布式架构过程中, 把功能分散到节点的过程。在这个过程中, 每种类型的独立节点将会被指定一定的用途, 例如充当用于管理 TCP/IP 连接池的客户端网关或者提供认证或支付方面的服务等。针对每类节点, 我们定义了一种方式来指定接口, 并定义了每个节点所需的状态和数据。我们在这章的末尾描述了常见的分布式架构模式以及将其相互连接所涉及的各种网络协议。

18 当你的分布式架构已经就绪后, 你还需要做出一些设计决策, 它们影响到容错性、适应性、可靠性和可用性。你已经了解了各种节点中需要哪些数据以及涉及哪些状态, 但问题是怎么在把这些节点分散开的同时保持其一致性呢? 你是要使用全共享 (share-everything)、部分共享 (share-something) 还是无共享 (share-nothing) 的方案, 以及当你采用强一致、因果一致、最终一致时需要面临何种取舍? 在第 14 章, 我们探讨了各种你可以采取的方案, 包括一些重试策略, 这些策略在碰到由于进程、节点或网络故障——例如网络服务供应商过载——而引发的请求超时时会有用。

按理说通过添加硬件来实现系统水平伸缩应该是可行的，可惜实际情况是，不当的设计会影响到水平伸缩性。在第 15 章，我们讲解了数据共享策略、一致性模型、重试策略带来的影响。我们介绍了容量规划方面的内容，包括常规负载、峰值负载、压力测试等方面，你必须保证系统的行为维持可预测，特别是在重负载情况下，周边硬件、软件、基础设施又出现失效时也是一样。

一旦设计好伸缩性策略和可用性策略，你就需要处理监控了。如果想要达到 5 个 9 的在线时间（uptime），你不仅需要知道此刻正在发生什么，还要能够快速确定已经发生了什么，为什么会发生。我们以第 16 章作为本书的结尾，看看监控功能是如何实现抢占式支持（preemptive support）以及事后调试的。

监控聚焦于指标（metric）、告警（alarm）和日志（log）三个方面。第 16 章讨论了对系统和业务进行测量的重要性。系统测量的例子包括节点消耗的内存、进程消息队列的长度、磁盘利用率等。将这些与业务度量相结合，例如尝试登录成功或失败的次数、每秒消息吞吐量、会话持续时间等，可以完整地看到你的业务逻辑是如何对系统资源产生影响的。

告警是对度量的补充，你用它来检测并报告异常情况，允许系统采取动作应对或者如果需要人类干预的话提醒运维人员。告警可以在系统即将用完磁盘空间时（触发自动调用脚本来压缩或删除日志）或者大量消息投递失败时（此时需要人类介入进行连接故障检修）。先发支持最好的状态，就是在问题尚未恶化前检测到并解决掉——这也是达成高可用性的必不可少的关键。如果你无法实时观察到正在发生的情况，想要在问题恶化前就解决是极难的。

最后，将重大事件记录到系统里有助于在系统崩溃状态丢失后进行故障检修，这样你才能在数以百万的客户服务查询处理过程中找到某个特定请求的调用流，例如查出特定的付款记录。

监控就绪后，你就已经架构好了不但可伸缩，而且具备弹性和高可用性的系统。阅读快乐！我们希望你喜欢这本书，就像我们喜欢写这本书一样。

19

Erlang 简介

本书假定你对 Erlang 的基础知识有一定的了解，获得这些知识的最好方式是实践，以及阅读一些介绍 Erlang 的优秀书籍（包括两本 O'Reilly 出版的书，参见本章最后的“总结”一节）。不过出于快速温习的目的，本章为你提供了一些 Erlang 重要概念的概览。对于其中部分知识我们会重点讲解，因为后面学习 OTP 时会用到。

递归与模式匹配

Erlang 程序员使用“递归”来迭代或重复某种行为。通过这种方式，还可以以间歇性的活动维持进程存活。我们的第一个例子展示了如何计算正数的阶乘：

```
-module(ex1).  
-export([factorial/1]).
```

```
factorial(0) ->  
    1;  
factorial(N) when N > 0 ->  
    N * factorial(N-1).
```

我们把这个函数叫作 `factorial` 并且指定它接受一个参数（`factorial/1`）。尾部的 `/1` 是函数的元数，简单明了地点明了函数接受的参数个数——在这个例子里是 1。

如果我们把整数 0 作为参数传给此函数，将会与第一个分句（`clause`）相匹配，返回 1。任何大于 0 的整数都会被绑定到变量 `N` 上，返回 `N` 与 `factorial(N-1)` 的积。迭代过程会持续直到与作为基例（`base case`）的函数分句匹配为止。“基例”是使得递归停止的分句。

22 如果我们以负数为参数调用 `factorial/1`，调用会失败，因为没有匹配的分句。但我们无须因为调用者可能传入非整数参数导致问题而不安，这属于 Erlang 的一个原则，之后会讨论。

Erlang 定义包含在模块里，而模块又以同样的名字存为后缀为 *erl* 的文件。因此，前面这个模块的文件名应该叫 *ex1.erl*。Erlang 程序可以放在 Erlang shell 中执行。在你的 UNIX shell 中调用 *erl* 命令或者通过双击 Erlang 图标，就能打开 Erlang shell。请确保在与你的源代码相同的目录处启动 Erlang shell。常规过程类似下面这样：

```
$ erl                                % 交互过程的注释都会以这种形式给出
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V7.2 (abort with ^G)
1> c(ex1).
{ok,ex1}
2> ex1:factorial(3).
6
3> ex1:factorial(-3).
** exception error: no function clause matching
    ex1:factorial(-3) (ex1.erl, line 4)
4> factorial(2).
** exception error: undefined shell command factorial/1
5> q().
ok
$
```

第 1 条命令，我们编译了 Erlang 文件。紧接着的第 2 条命令，我们以全限定形式调用了函数，所谓全限定形式，就是把模块名放到函数名之前的形式，这样可以从该模块之外对该模块之内的函数进行调用。第 3 条命令所执行的调用失败了，出现了函数分句错误，是因为没有分句匹配负数。调用 shell 命令 *q()* 退出 shell 前，我们在第 4 条命令处调用了本地函数 *factorial(2)*，出现错误是因为缺少相应的模块名。

递归的用处绝非仅仅做一些简单的值计算，我们完全可以用这种风格来写命令式程序。下面的程序打印出一个列表中的各个元素，并以 Tab 分隔。和前一个例子一样，函数由两个分句构成，每个分句都有由箭头符号 (*->*) 分隔的头和体。在头部我们看到函数使用了一个模式，通过参数调用函数时，碰到的第一个其模式能与参数匹配的子句会被选中。在这个例子里，*[]* 匹配空列表，而 *[X|Xs]* 匹配一个非空列表。*[X|Xs]* 语法的含义是把列表中的第一个元素（也称为表头）指派给 *X*，然后把列表的其余部分（也称为表尾）指派给 *Xs*（如果你还没注意到，提醒你一下，Erlang 中的变量，如 *X*、*Xs* 和 *N* 都以大写字母开头）。

```
-module(ex2).
-export([print_all/1]).
```

```
print_all([]) ->
```

```

    io:format("~n");
print_all([X|Xs]) ->
    io:format("~p\t", [X]),
    print_all(Xs).

```

上述代码的效果是从列表中打印各个元素，顺序与元素在列表中的位置相同，并在每个元素之后带一个 Tab (\t)。幸亏有基例存在，一旦列表为空（匹配 []）时就会运行它，进而打印一个换行符 (~n)。不同于先前 ex1:factorial/1 一例中的情况，本例中的递归属于尾递归。Erlang 程序中使用尾递归来实现循环。一个函数要能被称作尾递归，必须满足其对自身的递归调用是该函数分句中执行的最后一个表达式。我们可以把这一末尾的调用想象成“跳”回了函数起始处，以新的参数调用。尾递归函数支持 last-call 优化，这样就不需要每次迭代都添加新的栈帧，进而函数便可以以恒量的内存空间执行，避免了栈溢出问题——Erlang 中栈溢出表现为虚拟机耗尽内存。

如果你有命令式编程背景，我们把上面例子中的函数改写一下，不使用多个单独的子句，而是使用 case 表达式，这样也许可以让你理解起来更容易一些^{注1}：

```

all_print(Ys) ->
    case Ys of
        [] ->
            io:format("~n");
        [X|Xs] ->
            io:format("~p\t", [X]),
            all_print(Xs)
    end.

```

当你对上述打印函数进行测试时，请注意换行符后打印出的 ok。这是因为每个 Erlang 函数都必须返回一个值，这个值是由最后一个执行的表达式返回的。在我们的例子里，最后执行的表达式是 io:format("~n")。输出换行符源于该函数的副作用，而输出 ok 则是因为 shell 将该函数返回的值给打印了出来：

```

1> c(ex2).
{ok,ex2}
2> ex2:print_all([one,two,three]).
one    two    three
ok
3> Val = io:format("~n").
ok
4> Val.
ok

```

注1 但是这样比较丑，因为我们在函数头部使用的是 case 表达式而非模式匹配。

在我们的例子里，参数扮演了可变变量的角色，它们的值随每次调用而变化。Erlang 变量是单赋值的，一旦变量与值绑定，就不能再改变该变量。在递归函数中，作为函数迭代时，包括函数参数在内的全部同名变量，每次都会新建。通过下面的例子我们可以观察单赋值变量的行为：

```
1> A = 3.
3
2> A = 2+1.
3
3> A = 3+1.
** exception error: no match of right hand side value 4
```

第一行命令，我们赋值给一个未绑定变量的操作成功了。第二行命令，我们把一个已赋值的变量与其值做模式匹配。第三行命令也是模式匹配，但失败了，因为右侧值和 A 当前的值不一样。

在 Erlang 中还可以针对二进制数据进行模式匹配，匹配在比特级别进行。这个功能在对帧解码和处理网络协议栈时非常强大，而且结构清晰效率高。比如，寥寥数行就能完成解码 IPv4 包：

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

handle(Dgram) ->
  DgramSize = byte_size(Dgram),
  <<?IP_VERSION:4, HLen:4, SvcType:8, TotLen:16, ID:16, ...,
    Flgs:3, FragOff:13, TTL:8, Proto:8, HdrChkSum:16, ...,
    SrcIP:32, DestIP:32, Body/binary>> = Dgram,
  if
    (HLen >= 5) and (4*HLen <= DgramSize) ->
      OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
      <<Opts:OptsLen/binary, Data/binary>> = Body,
      ...
  end.
```

我们首先确定 Dgram 的尺寸（字节数）——Dgram 是一个变量，持有从网络套接字收到的一个二进制数据包。接着，针对 Dgram 进行模式匹配，抽取出其字段，该模式匹配赋值的左值定义了一个 Erlang 二进制数据，<<和>>的中间是它的一组字段。二进制里的省略号(...)不是合法的 Erlang 代码，只是表示此处省略了一些字段，意在使例子简单明了。大多数字段后跟着的数字表示的是该字段所占的比特数（如果该字段是二进制类型则为字节数）。比如，Flgs:3 对应一个 3 比特的整数，值绑定到变量 Flgs 上。作为模式匹配时最后一个字段 Body 的尺寸我们还不知道，所以把它指定为一个字节长度未知的

二进制数；这一做法在变量 `Data` 处也用到了。如果模式匹配成功，仅此一句语句，就把 `Dgram` 数据包中的所有命名字段抽取出来了。最后，针对抽取出来的 `HLen` 字段用 `if` 语句进行判断，如果成立，就对 `Body` 进行模式匹配赋值，以二进制方式抽取出 `Opts`，其长度为 `OptsLen` 字节，并且把 `Body` 中余下的数据都绑定到 `Data` 变量。请留意 `OptsLen` 是怎样动态计算的。如果你曾用 `Java` 或 `C` 之类的命令式语言写过这种从网络包中抽取字段的代码，你自会体会到其妙处。

受函数式的影响

Erlang 受到了其他一些函数式编程语言的影响。函数式的一个原则是把函数作为第一类公民；可以把函数赋给变量，可以把函数作为复杂数据结构的一部分，可以把函数作为参数传递给另一个函数，或作为函数调用的结果返回。这种函数式的数据类型我们称其为匿名函数（anonymous function），或简称 *fun*。除此之外，Erlang 还提供了一些构造方式，允许你以“生成并测试”的方式定义列表——类似集合论中的概括（comprehension）。我们先从匿名函数——没有名字的，不是定义在 Erlang 模块层面的函数——开始吧。

玩转匿名函数

接收函数作为参数的函数称为高阶函数。此类函数的一个例子是 `filter`（过滤函数），其断言以返回 `true/false` 的匿名函数表达，作用于列表中的元素。`filter` 函数返回一个新列表，其中的元素都满足某种特质；换句话说，也就是那些经匿名函数判定后返回为 `true` 的元素。我们常用术语“断言”指代基于某种条件进行判断然后返回 `true` 或 `false` 的匿名函数。下面的例子是 `filter/2` 函数的一种可能的实现。

```
-module(ex3).  
-export([filter/2, is_even/1]).
```

```
filter(P,[]) -> [];  
filter(P,[X|Xs]) ->  
    case P(X) of  
        true ->  
            [X|filter(P,Xs)];  
        _ ->  
            filter(P,Xs)  
    end.
```

```
is_even(X) ->  
    X rem 2 == 0.
```

要使用 `filter` 函数，你需要传给它一个函数和一个列表。一种办法是使用 `fun` 表达式，

fun 表达式同时也是定义匿名函数的方式。在命令 2 中，如下所示，你可以看到用了一个匿名函数来测试参数中哪些是偶数。

```
2> ex3:filter(fun(X) -> X rem 2 == 0 end, [1,2,3,4]).
[2,4]
3> ex3:filter(fun ex3:is_even/1,[1,2,3,4]).
[2,4]
```

fun 不是只能用作匿名函数，也可以用它引用局部的或全局的函数定义。在命令 3 中，我们通过 fun ex3:is_even/1 描述了一个函数；其中给出了模块名、函数名，以及元数。分裂进程时，匿名函数可以作为其起始函数，匿名函数还可以作为消息在进程间传递，在下一个主题里我们会涉及进程。

如果你用的 Erlang/OTP 版本是 17.0 或更高，还有一种非匿名使用 fun 的方式，你可以给它设定一个名字。这个功能在 shell 里使用特别方便，因为用它可以轻松地定义递归的匿名函数。例如，可以在 shell 里这样实现与 ex3:filter/2 等价的函数：

```
4> F = fun Filter(_,[]) -> [];
4> Filter(P,[X|Xs]) -> case P(X) of true -> [X|Filter(P,Xs)];
4> false -> Filter(P,Xs) end end.
#Fun<erl_eval.36.90072148>
5> Filter(fun(X) -> X rem 2 == 0 end,[1,2,3,4]).
* 1: variable 'Filter' is unbound
6> F(fun(X) -> X rem 2 == 0 end,[1,2,3,4]).
[2,4]
```

我们通过在 fun 关键词后放上名字来命名递归函数 filter。注意，名字必须在两个函数分句中都出现：第一行的分句处理空列表情况；接下来两行处理非空列表情况。在第二个分句中有两个地方都递归地调用了 filter 函数来处理列表中的剩余元素。但是即使函数有名字 filter，我们还是把它赋值给 shell 变量 F，是因为 filter 这个名字只是在函数自身内局部有效，不能在此之外调用，第 5 行展示了这一点。在第 6 行，我们通过 F 调用则没问题，与预期一致。另外，由于 shell 变量和函数名是在不同的作用域内，我们可以使用 shell 变量 filter 而不是 F，这样一来在两个作用域内函数的名称都是一样的了。

列表推导：生成与测试

27

我们目前看到的大部分例子都和列表操作有关，主要使用递归函数和高阶函数。其实还有另一种方法——列表推导（list comprehension），它是一种能够生成元素并对生成的元素做测试的表达式。格式类似下面这样：

```
[Expression || Generators, Tests, Generators, Tests]
```

其中 Generator 部分的格式为：

```
X <- [2,3,5,7,11]
```

效果为把值 2、3、5、7、11 逐个绑定到变量 X。即从列表生成 (generate) 元素：符号 <- 相当于集合的“属于其元素 (element of)”符号 \in 。在这个例子里，X 被称为绑定变量 (bound variable)。这个例子中只有一个绑定变量，但其实列表推导可以由多个绑定变量和生成器构成，本节后面会给出一些例子。

Tests 是一些布尔表达式，它们针对值和绑定变量的每一种组合而求值。如果同一组内的全部 Tests 都返回 true，则从当前的绑定变量和它们的值上生成表达式。在列表推导中使用或不使用 Tests 都可以。列表推导构造作为一个整体产生一个对应的结果列表，其中的元素对应了绑定变量的所有能够满足测试条件的组合。

作为第一个例子，我们来把 filter/2 函数改写为列表推导形式：

```
filter(P,Xs) -> [ X || X<-Xs, P(X) ].
```

在这个列表推导中，第一个 X 是其表达式，X <- Xs 是其生成器，而 P(X) 则是测试。生成器生成的每个值都会进行测试，测试为 true 的才会进一步构成表达式。对于测试返回 false 的值则简单地被丢弃掉。可以直接在程序里使用列表推导（像上面 filter/2 例子里那样），或者在 Erlang shell 里使用：

```
1> [Element || Element <- [1,2,3,4], Element rem 2 == 0].
[2,4]
2> [Element || Element <- [1,2,3,4], ex3:is_even(Element)].
[2,4]
3> [Element || Element <- lists:seq(1,4), Element rem 2 == 0].
[2,4]
4> [io:format("~p~n",[Element]) || Element <- [one, two, three]].
one
two
three
[ok,ok,ok]
```

28 注意在 shell 命令 4 里，我们是怎么在使用列表推导的同时创建副作用的。表达式最后返回了列表 [ok, ok, ok]，其中包含的值是对元素调用 io:format/2 后的返回值。

接下来这组例子展示了使用多个生成器以及交叉使用多个生成器与测试的效果。第一个例子针对 X 的每一个值，Y 依次绑定到 3、4、5。第二个例子，Y 的值取决于 X 取何值（这说明表达式先求 X 的值然后才求 Y 的值）。其余两个例子中对两个绑定变量执行了测试。




```

5> [ {X,Y} || X <- [1,2], Y <- [3,4,5] ].
[{1,3},{1,4},{1,5},{2,3},{2,4},{2,5}]
6> [ {X,Y} || X <- [1,2], Y <- [X+3,X+4,X+5] ].
[{1,4},{1,5},{1,6},{2,5},{2,6},{2,7}]
7> [ {X,Y} || X <- [1,2,3], X rem 2 /= 0, Y <- [X+3,X+4,X+5], (X+Y) rem 2 == 0 ].
[{1,5},{3,7}]
8> [ {X,Y} || X <- [1,2,3], X rem 2 /= 0, Y <- [X+3,X+4,X+5], (X+Y) rem 2 /= 0 ].
[{1,4},{1,6},{3,6},{3,8}]

```

给你留一个列表推导的思考题^{注1}吧，这个例子可以算是我们最喜欢的例子之一了，给定一个 8×8 的棋盘，将 N 个皇后放在上面，使得它们无法相互攻击的放法有多少种？在我们的例子里，`queens(N)` 返回皇后在棋盘底部 N 行可以选择的位置列表，其中每个元素是一个由（位于指定行的）列号构成的列表。为了找出可能的不同放法数，我们直接计算其排列数。注意，`--` 是列表差运算符，它与 `++` 相反，`++` 是把列表附加到另一个列表上。我们用了 `andalso` 而没用 `and`，因为当表达式求值得 `false` 时，它是一个短路运算符。

```

-module(queens).
-export([queens/1]).

queens(0) -> [[]];
queens(N) ->
    [[Row | Columns] || Columns <- queens(N-1),
     Row <- [1,2,3,4,5,6,7,8] -- Columns, % -- 运算符会返回列表间的差异部分
     safe(Row, Columns, 1)].

safe(_Row, [], _N) -> true;
safe(Row, [Column|Columns], N) ->
    (Row /= Column + N) andalso (Row /= Column - N) andalso
    safe(Row, Columns, (N+1)).

```

进程与消息传递

29

并发在 Erlang 编程模型中处于核心地位。进程是轻量的，意味着创建它们只需花费微不足道的时间和极少的内存。进程间不共享内存，而是通过消息传递来通信。消息从发送进程的栈上复制到接收进程的堆上。由于多个进程并发地在独立的内存空间执行，这些内存空间可以独立地进行垃圾回收，这给 Erlang 程序带来了良好的可预测的软实时属性，并且即使处于重负载下也能维持这一点。数百万个进程可以同时运行在一个 VM 里，各自处理独立的任务，当出现异常时进程就无法继续了，但因为进程间不共享内存，错误是隔离的。这允许其他执行不相关或不受此影响的任务的进程继续运行，整个程序能够自己恢复正常。

注1 你应当感谢这个例子。当还是学生时，本书作者中的某一位曾经花费了两个通宵只为解决这个问题，直到 Joe Armstrong 告诉他其实只需 4 行代码就能解决。



那么,这是怎么实现的呢? 进程是通过 `spawn(Mod,Func,Args)` 函数(或其变体)创建的,它是一个 BIF 函数。`spawn` 调用后会返回一个进程标识符,通常称为 `pid`。`pid` 是用来进行消息发送的,并且其自身也能放到消息里,这样可便于其他进程反向通信。如我们在图 2-1 里所见,进程以函数 `Func` 开始执行,该函数定义在模块 `Mod` 里,并且传递了参数列表 `Args`。

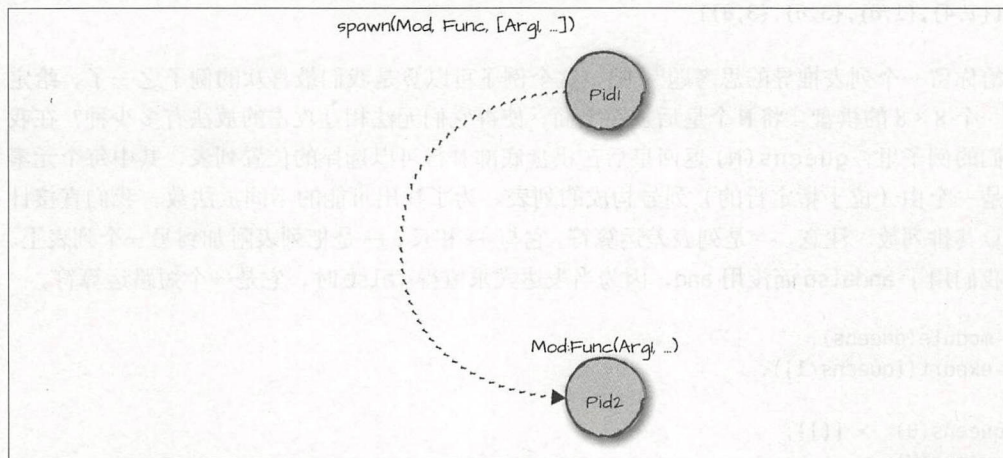


图 2-1: 分裂一个新进程

下面的例子是一个“回声”进程,展示了上述基础知识。`go/0` 函数的第一个动作是分裂出一个进程执行 `loop/0` 函数,然后它通过发送和接收消息与那个进程通信。`loop/0` 函数接收消息,并且根据消息的格式,要么做出回应(并继续循环),要么终止运行。为了让这个行为能不断循环进行,该函数是尾递归的,确保其执行只需常量内存空间。

30

对分裂出来执行 `loop/0` 的进程,它的 `pid` 我们是知道的,但当我们给它发送一条消息后,它怎么反过来与我们通信呢? 我们得把自己的 `pid` 发给它。通过使用 `self()` 这一内置函数可以知道我们自己的 `pid`:

```
-module(echo).
-export([go/0, loop/0]).

go() ->
    Pid = spawn(echo, loop, []),
    Pid ! {self(), hello},
    receive
        {Pid, Msg} ->
            io:format("~w~n", [Msg])
    end,
    Pid ! stop.
```




```

loop() ->
  receive
    {From, Msg} ->
      From ! {self(), Msg},
      loop();
    stop ->
      ok
  end.

```

在 echo 这个例子里，go/0 函数首先分裂出一个新进程执行 echo:loop/0 函数，并把返回的 pid 存在了变量 Pid 里。然后向 Pid 对应的进程发送了一条消息，该消息中包含发送方的 pid 以及一个原子 hello，发送方的 pid 是通过使用 self() 内置函数查到的。做完这些后，go/0 等待接收一条消息，其形式为一对元素，第一个元素是一个 pid 并且其值对应那个执行 loop/0 的进程；当这样的消息抵达时，go/0 打印出此消息的第二个元素，退出 receive 表达式，然后发送一条 stop 消息给 pid，结束。

echo:loop/0 函数首先等待消息。如果它收到一条由一对元素构成的消息——第一部分为来源 pid（存到 From 里），第二部分是一条消息（存到 Msg 里）——就回发一条消息，其中包含自己的 pid 以及从 From 进程那里收到的消息 Msg，然后递归调用自身。如果不是这样而是收到了原子 stop，loop/0 返回 ok。当 loop/0 停止，go/0 当初分裂出来运行 loop/0 的函数也就终止了，因为没有什么需要执行的代码了。

注意，当我们运行此程序时，go/0 调用返回了 stop。每个函数都会返回一个值，正是其最后一个求值的表达式的结果。在这里，最后一个表达式是 Pid!go，它返回我们发向 Pid 的消息：

```

1> c(echo).
{ok,echo}
2> echo:go().
hello
stop

```



模式匹配中的变量绑定

Erlang 中的模式匹配和其他语言中的有所不同，因为 Erlang 允许已经绑定的变量出现在模式匹配中。在 echo 例子的 go 函数中，变量 Pid 已经绑定为刚刚分裂的函数的 pid，这样一来，receive 表达式将只会接受第一部分指定为 pid 的消息；就此处的场景而言，实际上也就是来自该 pid 进程对应的消息。

如果收到的消息其第一部分不符合上述所述的内容，那么 receive 中的模式匹配也就不成功，进而 receive 将会继续阻塞直至来自 Pid 进程的消息出现。



Erlang 消息传递是异步的：发送消息到进程的表达式会立刻返回，并且看起来总是成功的——即使接收进程根本不存在。如果进程存在，则消息会被按顺序放在接收进程的邮箱里，这些消息会被 `receive` 表达式处理，按顺序执行模式匹配。消息接收是选择性的，因此不一定要按照消息抵达的顺序进行处理，而可以按照匹配顺序处理。每个 `receive` 子句通过使用模式匹配从邮箱读取消息来选择处理哪一条。

假设 `loop` 进程的邮箱依次收到了消息 `foo`、`stop` 和 `{pid, hello}`，`receive` 表达式将会尝试对第一条消息（即 `foo`）进行匹配，匹配时会依次测试每一种模式，如果不成功，这条消息就留在邮箱中。然后对第二条消息 `stop` 进行同样的尝试；这一次虽不匹配第一个模式但匹配第二个，因此结果是进程终止了，不会再执行任何代码。

上述语义意味着“我们可以以任何我们选择的顺序处理这些消息，不管消息是何时到达的”。代码类似下面：

```
receive
    message1 -> ...
end
receive
    message2 -> ...
end
```

其将会处理原子 `message1`，然后是原子 `message2`。没有这一功能，我们将不得不提前预料到各种消息抵达的所有可能的顺序，并处理这一切情形，这极大地增加了程序的复杂度。而借助选择性接收，我们要做的就是将它们留在邮箱里，等待之后取出。

32

多核、调度器和余量

在多核架构上进行系统伸缩时最大的挑战是顺序化代码和序列化操作。它们可能存在于你的程序中、你使用的库中、下层虚拟机中，甚至到处都是。内存锁的争用常是主要瓶颈，原因是多个线程都试着获得锁来访问和操作共享内存。Erlang 进程间不共享内存，移除了这一主要障碍，使其成为能够充分利用众核（many-core）计算机的理想语言。在单核机器上运行 Erlang 编写的程序，只要保证每个真正并发的活动都有对应的独立进程，那么你的系统就可以随着核心增加而扩展。能限制你的只有你的顺序代码以及 BEAM 虚拟机里的瓶颈——随着一个又一个新的版本产生，这种瓶颈会被不断优化或消除。

针对每一个核，BEAM 虚拟机启动一个线程，其上运行一个调度器。每个调度器负责一组进程，在任一时刻，不同核心上的不同调度器都有一个进程在并行执行。



未暂停并且已经准备好执行的进程被放到调度器的运行队列里。虚拟机还启动了一个单独的线程池用于驱动和进行文件输入输出，它们可以独立操作而不至于阻塞任何调度器线程。在虚拟机启动时，你可以限制线程和调度器的数量，并指定你是想把调度器绑定到核心，还是允许调度器从一个核心迁移到另一个核心。默认情况下，调度器是不与核心绑定的，因为绑定可能适得其反，减慢系统在特定架构下的进行速度。不过，在另一些情况下，绑定也可能带来提速。对你的系统在这两种情况下的性能都进行基准度量。在第 11 章的“参数和标志”小节我们讲到了如何设定启动标志和参数，第 15 章则讲解了基准度量。

如果系统运行在满载状态，调度器将通过在全部进程间维持尽可能平均的 CPU 时间分配来保证整个系统的软实时属性。BEAM 虚拟机想要避免这样的情形——1 个有 10 进程的运行队列获得了将近 2 倍的 CPU 时间；与另一个有 20 个进程的运行队列所获得的相差无几。实现这一点的关键是允许进程在运行队列间迁移，使得各个调度器的运行队列大小均衡。但如果系统不处于满载状态，虚拟机将迁移进程使得占用的核减少，然后暂停用不到的调度器线程。这样一些核心可以被关闭并置于节能模式，之后虚拟机负载增加后再唤醒它们。

调度器抢占进程是根据该进程工作量决定的。这种估计方法称为余量（reduction）计数。当进程被抢占的时候，其被停止运行并放到运行队列尾部，允许队列中此时位于第一的进程执行。函数调用和 BIF 调用被指定一个或多个余量，遵循昂贵调用的余量比廉价调用的余量高的理论。每个进程都被允许执行一个预定数量的余量，然后才被抢占，把执行让给处于运行队列头部的进程。对于进程被暂停前允许执行的余量数量，及每条指令对应的余量数都故意没有提供对应的文档可供参考，因为这些量随着版本、更迭及硬件架构差异都可能不同，我们反对依赖这些量做出不成熟的优化。

调度器平衡、余量，以及每进程（per-process）垃圾回收使得，即使在峰值或过载（extended load）状态下，也能通过最大化公平性及确保无进程饥饿来保证 BEAM 虚拟机具有可预测性及软实时属性。其他不是运行在 BEAM 上的编程语言和框架则没提供抢占式多任务。应用程序活动是不允许被阻塞的，因为事件循环过程需要频繁运行，使得各种事件能够投递到目标。如果应用程序被阻塞，将影响到各个部分，然而在 Erlang 里，唯一能阻塞调度器（以及其运行队列中的全部进程）的方法就是用 C 语言愚蠢地或故意地实现一个不正常的原生函数（NIF）或驱动。因此，缺少抢占式多任务将影响到系统的软实时属性，并且要么得靠进程相互配合来实现抢占，要么把抢占建立在特定的操作上，而不是根据操作量和开销。说到这里，那种在共享内存式架构上“暂停一切”（stop the world）的垃圾回收器也是一大问题，它为了确定哪些对象还在使用哪些已经可以释放会强制所有线程进行同步。BEAM 没有类似它们的东西，BEAM 也没有类似它们的尴尬。



不怕出错!

在本章开始的“递归与模式匹配”一节中我们看过 `factorial` 的例子，并了解到当传递负数给该函数时会引发异常。如果传给 `factorial` 函数的连数字都不是，而是其他东西的话同样会引发异常，比如下面例子里传入了原子 `zero`。

```
1> ex1:factorial(zero).
** exception error: bad argument in an arithmetic expression
in function ex1:factorial/1
```

应对这一问题的一种方式防御式编程，通过使用一条“对应一切”的分句来明确指明负数以及非数字参数等其他情形的处理方式。

```
factorial(0) ->
    1;
factorial(N) when N > 0, is_integer(N) ->
    N * factorial(N-1);
factorial(_) ->
    {error,bad_argument}.
```

34 这样做会使该函数的每一个调用者不仅要处理正常返回值，如 `120=factorial(5)`，还必须处理非正常返回值——格式为 `{error, bad_argument}`。如果我们这么做，使用这一函数的人必须理解这一错误模型，并提供处理办法，这就把正确时的计算代码与错误时的应对代码混在了一起。问题是如果你不清楚这些错误的含义以及数据出了什么问题，你又如何能够知道应该如何处理呢？

Erlang 设计哲学说道：“任其出错”。函数、进程或其他运行的实体只处理正确时的情形，而出错的情况让系统（特别是这样生而容错的系统）去处理吧！在顺序代码中处理出错的方式之一是使用基于 `try-catch` 构造的异常处理。使用如下定义：

```
factorial(0) ->
    1;
factorial(N) when N > 0, is_integer(N) ->
    N * factorial(N-1).
```

我们可通过实践来了解 `try-catch` 构造：

```
2> ex1:factorial(zero).
** exception error: no function clause matching ex1:factorial(zero)
3> try ex1:factorial(zero) catch Type:Error -> {Type, Error} end.
{error,function_clause}
4> try ex1:factorial(-2) catch Type:Error -> {Type, Error} end.
{error,function_clause}
```




```

5> try ex1:factorial(-2) catch error:Error2 -> {error, Error2} end.
{error,function_clause}
6> try ex1:factorial(-2) catch error:Error3 -> {error, Error3};
6>                               exit:Reason -> {exit, Reason} end.
{error,function_clause}

```

try-catch 构造让使用者可以通过分句匹配不同类型的异常，分别进行处理。在这个例子里，我们匹配了由于模式匹配失败而产生的 error 异常。另外还有 exit 和 throw 异常，前者是由于进程调用了 exit 内置函数产生的，后者则是在某处通过 throw 表达式生成的。

用于监督的链接与监视器

常规 Erlang 系统内有许多进程同时运行，进程间可能相互依赖。存在依赖的情况下我们的“任其出错”哲学怎么办？假设进程 A 与进程 B 和进程 C 交互（参见图 2-2）；这些进程相互依赖，因此如果 A 出错，B 和 C 也无法继续正常运行。A 的出错必须被检测到，然后 B 和 C 都需要被终止，最后才可以把它们全部重启。在这一节里，我们介绍达成这一切的机制，名为“进程链接”“退出信号”和“监视”。这些简单的构造能让我们打造出带有复杂监督策略的库，允许我们管理那些随时可能出错的进程。

35

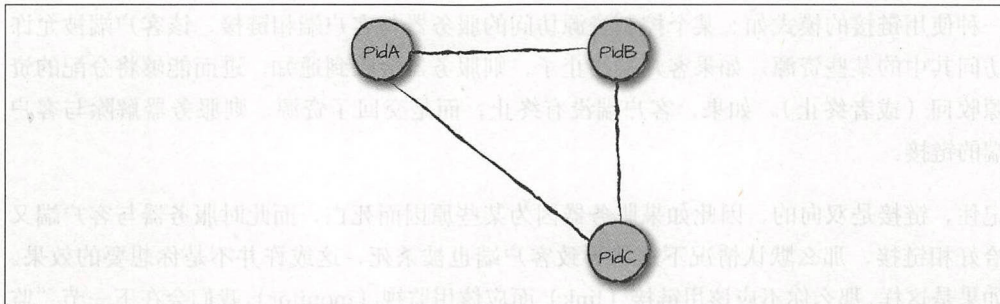


图 2-2: 进程间相互依赖

链接

在进程 A 内调用 link (Pid) 便可在进程 A 与 Pid 之间创建一个双向的链接。调用 spawn_link/3 的效果和先调用 spawn/3 紧接着调用 link/1 的效果一样，差别在于，这一切会自动执行，并且消除了由于进程可能在 spawn 和 link 两步中间终止而产生的竞态条件。调用 unlink (Pid) 可以移除调用进程与 Pid 之间的链接。

链接机制的关键是它必须与 Erlang 的消息传递机制正交，但又需要使用它来实现这一切。如果两个 Erlang 进程相链接，则其中之一终止时，将发送 exit 信号给另一方，使其也终



止。终止的进程会依次发送 `exit` 信号给所有与其链接的进程，这使得 `exit` 信号在系统内层层蔓延。图 2-3 展示了这一过程，注意，`PidC` 的终止可以是因为 `PidA` 发来的 `exit` 信号，也可以是因为 `PidB` 发来的 `exit` 信号，具体看哪一个先抵达，但无论先后结果都是一样的。这套机制的强大之处在于，它允许你修改上述默认行为，使得系统设计者能够很好地控制进程的终止过程。我们现在就来进一步了解一下。

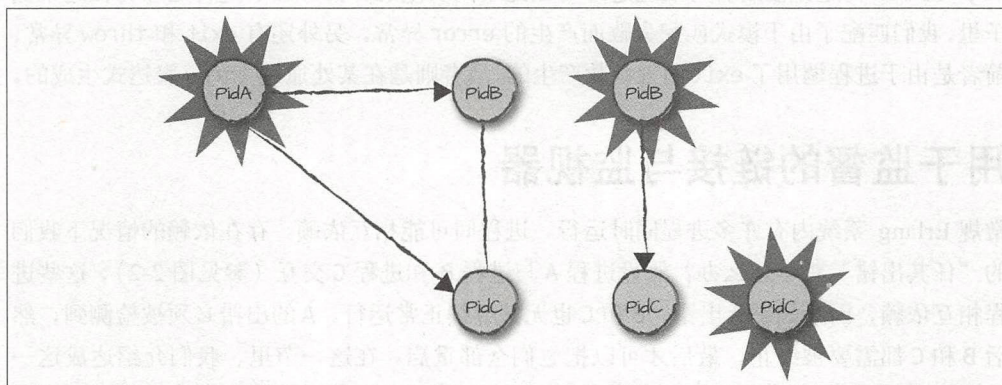


图 2-3: `exit` 信号在相互链接的进程间蔓延

36 一种使用链接的模式如：某个控制资源访问的服务器与客户端相链接，该客户端被允许访问其中的某些资源。如果客户端终止了，则服务器会收到通知，进而能够将分配的资源收回（或者终止）。如果，客户端没有终止，而是交回了资源，则服务器解除与客户端的链接。

记住，链接是双向的，因此如果服务器因为某些原因而死亡，而此时服务器与客户端又恰好相链接，那么默认情况下这将导致客户端也被杀死，这或许并不是你想要的效果。如果是这样，那么你不应该用链接（link）而应该用监视（monitor），我们会在下一节“监视器”中进行介绍。

通过调用 `process_flag(trap_exit, true)` 可以捕捉 `exit` 信号。这样做会使 `exit` 信号转变成一条格式为 `{'EXIT', Pid, Reason}` 的消息，其中 `Pid` 是死亡进程的标识符，而 `Reason` 则是其终止的原因。这些消息会存储在接收方的邮箱里，然后与其他消息一样接受处理。若进程捕捉 `exit` 信号，则不会再把 `exit` 蔓延给与其相链接的其他进程。

为什么进程会退出？这可能是两类原因导致的。一种情况是由于进程已经没有更多的代码需要执行了，这时它会正常退出。蔓延给外部的退出原因 `Reason` 将会是原子 `normal`。另一种情况则是非正常终止，例如出现运行时错误、收到 `exit` 信号而自己又不捕捉，以及主动调用了 `exit BIF` 等。当以单个参数调用时，`exit(Reason)` 将会终止调用此函数的进程，将终止原因 `Reason` 蔓延给与该进程相链接的其他进程。当 `exit BIF` 是以两个



参数调用时, `exit(Pid, Reason)` 则会发送 `exit` 信号给 `Pid` 进程, 并且原因为 `Reason`。这样做的效果与发起调用的进程已经由于原因 `Reason` 而终止一样。

就如我们在本节开头说过的那样, 用户可以控制系统中终止过程的蔓延。具体的选项在表 2-1 中做了汇总, 其最终行为根据进程是否设置了 `trap_exit` 标志而有所变化。

37

表 2-1: 蔓延语义 (Propagation semantics)

原因	捕捉 <code>exit</code> 时	不捕捉 <code>exit</code> 时
<code>normal</code>	接收到 <code>{'EXIT', Pid, normal}</code>	什么也不发生
<code>kill</code>	以 <code>killed</code> 为原因终止	终止, 原因为 <code>killed</code>
<code>Other</code>	接收到 <code>{'EXIT', Pid, Other}</code>	终止, 原因为 <code>Other</code>

表格中的第二列, 对于捕捉 `exit` 信号的进程, 当某个相链接的进程正常或者非正常终止时, 它会收到一条 `'EXIT'` 消息。如果终止原因是 `kill`, 则允许一个进程强制其他进程与它一起退出。这意味着存在一种机制可用于杀死任意进程——即使该进程捕捉 `exit`; 请注意进程被杀死后的终止原因是 `killed` 而不是 `kill`, 这保障了无条件终止不会向外蔓延。如果进程不捕捉 `exit` 信号, 则当与其相链接的进程正常终止时, 什么也不会发生。而如果是非正常终止, 则导致该进程也终止。

监视器

监视器 (monitor) 提供了另一种选择, 进程可以借助它单向式地观察另一些进程的终止。监视器与链接的不同体现在如下方面:

- 监视器的创建是以进程 A 调用 `erlang:monitor(process,B)` 开始的, 其中原子 `process` 指明了我们监视的目标是一个进程, 并且 `B` 是一个 `pid` 或已注册的名称。这使得 A 监视 B。
- 各个监视器通过 Erlang 引用 (reference) 相互区分, 该引用是在调用 `erlang:monitor/2` 的时候返回的, 具有唯一性。A 可以设置多个对 B 的监视器, 每一个都通过不同的引用区分。
- 监视器是单向的, 而非双向的: 如果进程 A 监视了进程 B, 则不代表进程 B 也监视了进程 A。
- 当被监视的进程终止时, 一条格式为 `{'DOWN', Reference, process, Pid, Reason}` 的消息会被发给监视此进程的进程。其中不仅包含终止进程的 `Pid` 以及终止原因 `Reason`, 还包含一个对监视器的引用 `Reference` 以及一个原子 `process`——告诉我们监视的目标是一个进程。
- 调用 `erlang:demonitor(Reference)` 可以移除监视器。调用时传入第二个参数 `erlang:demonitor(Reference,[flush])` 可以让该监视进程邮箱中所有与



Reference 对应的 {'DOWN', Reference, process, Pid, Reason} 消息都被冲刷掉。

- 尝试监视一个不存在的进程会导致收到一条 {'DOWN', Reference, process, Pid, Reason} 消息，其 Reason 为 noproc，这和链接到不存在的进程时所发生的不同，那会导致发起链接的进程终止。
- 如果受监视的进程终止，那些监视它并且没有捕捉 exit 信号的进程并不会终止。



reference (引用) 在 Erlang 中主要用于标记消息、监视器以及其他请求数据，使得我们能区分各个实体。在创建一个监视器的同时，会创建一个对应的 reference。但如果只是想创建 reference，则直接调用 `make_ref/0` BIF 即可。每一个 reference 都具有唯一性，即使在多节点 Erlang 系统中这一特性也成立，因为这是刻意设计而成的。reference 之间可以相互比较相等性，模式匹配时也能使用，基于此你能够判定消息是否来自某个特定的进程，或者判定是否属于某通信协议中的回复消息。

我们可以像这样试用一下 `monitor/2` 和 `exit/2`，结果无须多言，一目了然：

```
1> Pid = spawn(echo, loop, []).
<0.34.0>
2> erlang:monitor(process, Pid).
#Ref<0.0.0.34>
3> exit(Pid, kill).
true
4> flush().
Shell got {'DOWN', #Ref<0.0.0.34>, process, <0.34.0>, killed}
ok
```

记录

Erlang 中的 tuple (元组) 提供了一种分组相关项的方式，和列表不同，这种方式提供了便捷的访问任意元素的方式，不需要像列表那样借助 `element/2` BIF。不过在实践中，用元组管理的项数不宜超过五六项。项数超过这一级别维护起来就会很头疼，因为你必须当心代码中使用到元组的各处的位置匹配是否正确，更糟的是，如果你使用 `element/2` 函数以纯数字方式来定位字段，那么就更容易犯错了。对长元组做模式匹配是一件相当乏味的事，因为你必须确保你的匹配模式中长度对应正确，且每个变量放置的位置也对应正确。最糟糕的是，如果你必须在某个元组中添加或者删除字段，则必须找出代码中所有用到这个元组的地方，然后逐一修改。

record (记录) 解决了 tuple (元组) 在这些方面存在的不足，它依然是一种元素式的集合，但允许使用名字来访问其中的子段。下面是在 Erlang/OTP 的 `inet` 模块中使用了记录的一个例子，可以看到我们是如何访问其中的 TCP/IP 信息的：




```
-record(hostent,
{
    h_name           % 主机的正式 (official) 名称
    h_aliases = []   % 别名列表
    h_addrtype       % 主机地址类型
    h_length         % 地址长度
    h_addr_list = [] % 来自名字服务器 (name server) 的地址列表
}).
```

-record 指令用于定义记录，其中第一个参数指定了记录的名字。第二个参数，是一个由原子 (atom) 构成的元组，定义了记录中有哪些字段。字段可以指定默认值，如 h_aliases 和 h_addr_list 字段所示，二者都使用空列表作为默认值。对于没有明确指定默认值的字段则其默认值为原子 undefined。

赋值、模式匹配、函数参数等地方都可以使用记录，这一点和元组一样。但和元组不同的是，记录的字段是通过名字访问的，并且如果当前代码中没有用到某个字段，则该字段可以完全不出现在当前代码中。例如，下面模块中的 type/1 函数只需要访问到 hostent 记录中的 h_addrtype 字段：

```
-module(addr).
-export([type/1]).

-include_lib("kernel/include/inet.hrl").

type(Addr) ->
    {ok, HostEnt} = inet:gethostbyaddr(Addr),
    HostEnt#hostent.h_addrtype.
```

首先请注意，为了能使用某个记录，我们必须能访问到其定义。此处的 -include_lib(...) 指令把 inet.hrl 文件从 kernel 应用中引入进来，就是因为该文件中包含了 hostent 记录的定义。例子中的最后一行，HostEnt 变量后跟 # 符号和记录名 hostent，作用使其能被以记录的方式访问。在记录名 hostent 后，我们直接以名字方式指明了想访问的字段是 h_addrtype。这样我们就能读取出存储在该字段中的值，并作为 type/1 函数的返回值返回。

```
1> c(addr).
{ok,addr}
2> addr:type("127.0.0.1").
inet
3> addr:type("::1").
inet6
```



实现 `type/1` 函数还有一种办法是针对 `inet:gethostbyaddr/1` 函数的返回值做模式匹配，获得 `h_addrtype` 字段：

```
type(Addr) ->
    {ok, #hostent{h_addrtype=AddrType}} = inet:gethostbyaddr(Addr),
    AddrType.
```

此处，模式匹配中的 `AddrType` 变量捕获了 `h_addrtype` 字段的值。这一形式涉及记录的模式匹配很常见，并且常用在函数头部，用于抽取出感兴趣的字段值并存储到本地变量中。如你所见，此种形式比前面例子中提到的那种字段访问语法更简练。

在创建记录的实例时，也只需按实际需要设定其中的部分字段：

```
hostent(Host, inet) ->
    #hostent{h_name=Host, h_addrtype=inet, h_length=4,
             h_addr_list=inet:getaddrs(Host, inet)}.
```

在这个例子里，`hostent/2` 函数返回一个 `hostent` 记录的实例，并且只设定了其中部分字段的值。其余没有在代码中显式设置其值的字段则根据记录定义自动设置为默认值。

记录实质上只是一种语法糖；在其糖衣之下，它们都是靠元组来实现的。要看清这一点，只需在 Erlang shell 中调用 `inet:gethostbyname/1` 函数：

```
1> inet:gethostbyname("oreilly.com").
{ok,{hostent,"oreilly.com",[],inet,4,
      [{208,201,239,101},{208,201,239,100}]}}
```

```
2> rr(inet).
[connect_opts,hostent,listen_opts,...]
3> inet:gethostbyname("oreilly.com").
{ok,#hostent{h_name = "oreilly.com",h_aliases = [],
             h_addrtype = inet,h_length = 4,
             h_addr_list = [{208,201,239,101},{208,201,239,100}]}}
```

在 shell 命令 1 中，我们调用 `gethostbyname/1` 获取了 `oreilly.com` 主机的地址信息。返回的结果是一个元组，其中第二个元素是一个 `hostent` 记录，但是 shell 却把它作为普通的元组显示出来——其中第一个元素正是记录的名字，而剩下的元素则是记录的各个字段，顺序与记录声明时一致。注意，记录实例中并没有包含记录声明时的那些字段名。要让记录实例显示为记录而不是元组，我们需要把记录的定义告知 shell。我们在 shell 命令 2 中使用 `rr` 命令就是出于此目的，它会根据参数读取记录定义，然后以列表方式返回读取到的定义（本例中我们使用省略号略去了大部分无关的项）。传递给 `rr` 命令的参数既可以是模块名，也可以是某个源文件或者头文件的名字，还可以是能够被 `filelib:wildcard/1,2` 函数接受的那类通配名。在 shell 命令 3 中，我们又一次获取了

oreilly.com 的地址信息，但是这一次 shell 是以记录格式输出的 hostent 值，包含了各个字段名。



正确的记录版本

当你改变了某个记录的定义后，需要极其小心地处理由此带来的记录版本不一致问题。你可能会忘了去重新编译某个用到该记录的模块（或者虽然编译了，使用的却是错误的记录版本），或者在 shell 里加载了错误的记录定义，或者把新版本的记录实例发给一个尚未升级还在使用老版本记录的进程。犯这些错误，运气好的话会在访问或操作某个不存在的字段时抛出异常，运气不好则什么提示也没有，只是默默赋值 / 返回了错误的字段。

映射组

Erlang 中的映射组 (map) 是一种键值集合类型，类似于其他编程语言中的字典和哈希类型。与记录相比，映射组有几个方面的不同：映射组属于内置类型，它的字段（或者说键值对）数量不会在编译期就固定下来，而且任意 Erlang 数据项 (term) 都可以做它的键，不仅限于原子。有些人以为映射组是用于替代记录的，但实践中这二者面向的是完全不同的场景，并且都很有用。记录的速度很快，所以当你拥有固定数量的字段时——并且它们在编译时便能确定下来——就用记录，而当你需要在运行时动态添加字段时则使用映射组。

创建和操作映射组很直接，如下所示：

```
1> EmptyMap = #{}.
#{}
2> erlang:map_size(EmptyMap).
0
3> RelDates = #{ "R15B03-1" => {2012, 11, 28}, "R16B03" => {2013, 12, 11} }.
#{"R15B03-1" => {2012,11,28},"R16B03" => {2013,12,11}}
4> RelDates2 = RelDates#{ "17.0" => {2014, 4, 2}}.
#{"17.0" => {2014,4,2},
  "R15B03-1" => {2012,11,28},
  "R16B03" => {2013,12,11}}
5> RelDates3 = RelDates2#{ "17.0" := {2014, 4, 9}}.
#{"17.0" => {2014,4,9},
  "R15B03-1" => {2012,11,28},
  "R16B03" => {2013,12,11}}
6> #{ "R15B03-1" := Date } = RelDates3.
#{"17.0" => {2014,4,2},
  "R15B03-1" => {2012,11,28},
  "R16B03" => {2013,12,11}}
```

```
7> Date.  
{2012,11,28}
```

42 在 shell 命令 1 中，我们把一个空的映射组 `#{}` 绑定到了变量 `EmptyMap`，然后在 shell 命令 2 中使用 `erlang:map_size/1` 函数检查了它的尺寸。与预期相符，它的尺寸为 0，因为它不包含任何键值对。在 shell 命令 3 中，我们创建了一个新的映射组，其中包含多个键值对，每一个键都是 Erlang/OTP 的一个发行版本号，对应的值则是发行日期，使用了 `=>` 映射关联运算符（map association operator）。shell 命令 4 中使用已有的 `RelDates` 映射组为基础，添加了一个新的键值对创建了一个新的映射组 `RelDates2`。不幸的是，我们在 shell 命令 4 中设置的日期偏差了一个星期，需要把它改过来。shell 命令 5 展示如何使用 `:=` 映射组置值运算符（map set-value operator）更新发行日期。和 `=>` 运算符不同，`:=` 运算符会确保映射组中存在要更新的那个键，防止开发人员由于将键名拼写错误，导致意外地创建出新的键值对，而不是更新已有的键。最后，shell 命令 6 展示了如何把映射组用于模式匹配中，使得我们能够把与键 `"R15B03-1"` 对应的发行日期值捕获到变量 `Date` 中，在 shell 命令 7 里我们访问了该变量的值。注意，对映射组做模式匹配时必须使用 `:=` 置值运算符。

宏

Erlang 提供了宏（macro），这一功能是由 *epp*（Erlang preprocessor）实现的，每当要把源代码编译为 BEAM 代码之前都会调用它。可以定义常量宏，如下所示：

```
-define(ANSWER,42).  
-define(DOUBLE,2*).
```

还可以定义能接受参数的宏，如下所示：

```
-define(TWICE(F,X),F(F(X))).
```

从 `DOUBLE` 的定义里你可以看到，习惯上（当然也只是习惯上）宏名称使用大写。但实际上任何合法的 Erlang 语素（token）序列都可以，即使名称没能有效表达其实际含义也不算错误。

通过在宏名前带上 `?` 可以调用宏，如下所示：

```
test() -> ?TWICE(?DOUBLE,?ANSWER)
```

如果想清楚地看到宏是如何生效的，一个办法是在 shell 中编译时带上 `'P'` 标志：

```
c(<filename>,['P']),
```


这将会创建一个名为 *filename.P* 的文件，其中 `test/0` 的定义变成了：

```
test() -> 2 * (2 * 42).
```

另外，还可以把调用宏时传入的参数的原始文本记录下来。例如，如果我们定义这样一个宏：

```
-define(Assign(Var,Exp), Var=Exp,  
        io:format("~s = ~s -> ~p~n", [??Var, ??Exp, Var]) ).
```

那么 `?Assign(Var,Exp)` 将能在执行赋值操作 `Var = Exp` 的同时，输出一条诊断信息（这实际上是一种副作用）。例如：

```
test_assign() -> ?Assign(X, lists:sum([1,2,3])).
```

将会是这样：

```
1> macros:test_assign().  
X = lists : sum ( [ 1 , 2 , 3 ] ) -> 6  
ok
```

还可以借助标志（flag）分条件定义宏，例如：

```
-ifdef(debug).  
-define(Assign(Var,Exp), Var=Exp,  
        io:format("~s = ~s -> ~p~n", [??Var, ??Exp, Var]) ).  
-else.  
-define(Assign(Var,Exp), Var=Exp).  
-endif.
```

现在，如果你通过使用编译器标志 `{d,debug}` 启用了调试标志，则 `?Assign(Var,Exp)` 将会执行赋值操作，并输出诊断代码。相反，保持默认，不刻意设置调试标志，或者通过 `{u,debug}` 清除调试标志，都会使得程序只做赋值操作，而不会执行 `io` 表达式。

模块升级

动态类型的一个优点是能够在运行过程中升级代码，而无须关闭系统。某刻，你正运行着某个模块，而该模块的当前版本存在错误，为了解决这一问题，你无须终止进程，直接加载修复后的新版本即可，进程将会执行新版本，并且不会改变当前状态和变量（参见图 2-4）。这一做法不但适用于修复错误，也适用于升级和添加新功能。对于那些即使需要升级和维护也必须保证“5个9可用性”（99.999%）的系统来说，这是一个极其关

键的功能。

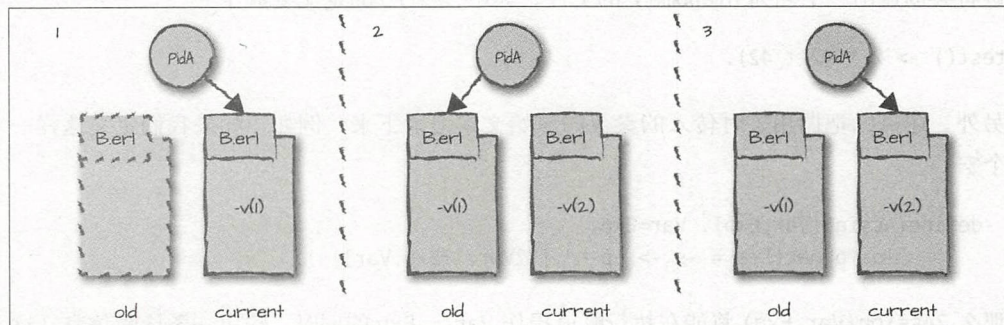


图 2-4：一次软件升级

44 在任意时刻，虚拟机内都可能同时存在同一模块的两个不同版本：老版本和当前版本。图 2-4 所示的第一个框中，PidA 正执行着模块 B 的当前版本。第二个框中，模块 B 的新代码被加载了进来，这要么是因为在 shell 中编译了该模块，要么是在 shell 外编译了但显式地进行了加载操作。在你加载了模块之后，PidA 链接到的仍然是与之前版本相同的模块 B——不过这时它已经可算作旧版本了。但当下次 PidA 再执行对模块 B 中函数的全限定调用（fully qualified call）时，将会触发一次版本检查操作，确保 PidA 运行最新版本的代码。（本章前面介绍过，全限定调用指的是带有模块名的函数调用。）如果进程运行的不是最新版本，指向代码的指针将会切换到当前的新版本，如第三个框中所示的那样。此过程不只是针对模块 B 中的某个函数有效，而是调用模块 B 中的任意函数时都会涉及。软件升级的实质基本就是这么一回事，但为了确保你能理解其中的细节，我们还是更进一步解释一下：

- 假设要升级的不是别的，恰好是运行中的进程的自己的循环（loop）代码。那么根据循环过程中函数调用的形式不同就会产生不同的影响。如果函数调用是全限定形式的——比如 `B:loop()`——那么下一次再调用时就会使用升级后的代码；否则，如果调用只是简单的 `loop()` 形式，那么进程将会继续使用老版本的代码。
- 同一时刻，系统只会至多持有同一模块的两个版本的代码，因此假如进程 p 还在执行着模块 B 的 `v(1)` 版本代码，此时另外两个版本 `v(2)`、`v(3)` 加载了进来：由于系统只能持有两个版本，因此最早版本 `v(1)` 将会被移除，这就使得一切依旧在那个老版本上循环的进程（比如 p）被无条件终止。
- 加载新代码有几个办法。一种是通过编译代码触发重加载，比如在 shell 里调用 `c(Module)` 或者调用 Erlang 函数 `compile:file(Module)`。另外，还可以在 shell 里执行 `l(Module)` 或者调用 `code:load_file(Module)` 来显式加载代码。普遍而言，当调用某个模块中的函数，而该模块尚未加载时也会触发加载。这将会使对应的

45

.beam 文件被加载，所以你必须保证该模块已经是编译好的，例如使用 `erlc` 命令行工具。注意，只使用 `erlc` 重编译一下模块是不会触发重加载的。

- 虽然新版本的代码加载后，老版本的代码就会被自动清除，但如果有需要，你也可以直接调用 `code:purge(Module)` 显式移除旧版本的代码（甚至在不加载新代码的情况下你也可以这么做）。这样做将使那些当前运行于旧代码之上的进程被终止，然后代码才被移除。如果有任何进程确实因此被终止了，则此调用返回 `true`，否则返回 `false`。调用 `code:soft_purge(Module)` 则只有当没有进程运行于旧代码之上时才会执行移除操作并返回 `true`，否则只返回 `false` 而不会终止任何进程。

ETS : Erlang 元素存储

列表作为一种重要的数据类型却只能被线性遍历，这使得它伸缩性不佳。如果你需要键值存储，并且查找时间为常量，或者需要能够以词典序遍历键，那么你可以用 ETS (Erlang Term Storage) 表。一张 ETS 表就是一些 Erlang 元组的集合，其统一取元组中某个位置处的元素作为键。

ETS 表又分为 4 种不同的类型：

集合 (set)

相同的 key-value 元组只可以出现一次。

包 (bag)

每种 key-value 元组组合只可以出现一次，但是同一个 key 可以出现多次。

重复包 (duplicate bag)

允许重复的元组。

有序集合 (ordered set)

和集合 (set) 的限制相同，但是可以按 key 的顺序访问各个元组。

访问有序集合 (ordered set) 类型中的元素时需要消耗表长度的对数级别的时间 $O(\log n)$ ，访问其余类型的元素则只需消耗常量级时间。

根据创建表 (`ets:new`) 时传入的选项不同，表可以具有如下所列的某一特点：

`public`

允许任何进程访问。

`private`

只有拥有该表的进程才能访问。

protected

允许任何进程读取，但是只有拥有该表的进程才能写入。

创建表时还可以通过 {keypos,N} 指定键取自哪个位置。这一功能在存储记录时很有用，因为它允许开发人员指定某个特定的字段作为键。默认情况下键取自位置 1。

正常情况下，程序要想访问表需要借助调用 new 时返回的表 ID，但创建表时是可以指定名字的，因此还可以通过名字来访问表。

每张表都会与创建它的进程相链接，并且当进程终止时该表会被自动删除。ETS 表仅存储在内存中，对于需要长期存储的表可以使用 DETS，它会存储在磁盘中（因此加了一个“D”，指“Disk”）。

下面展示了最基本的一些表操作：

```
1> TabId = ets:new(tab,[named_table]).
tab
2> ets:insert(tab,{haskell, lazy}).
true
3> ets:lookup(tab,haskell).
[{haskell,lazy}]
4> ets:insert(tab,{haskell, ghci}).
true
5> ets:lookup(tab,haskell).
[{haskell,ghci}]
6> ets:lookup(tab,racket).
[]
```

正如你看到的，默认的 ETS 表类型为 set，因此第 4 行插入的数据覆盖了第 2 行插入的数据，并且这张表的键对应的是第一个位置。同样值得注意的是，当查找某个键时，返回的是所有匹配该键的元组的列表。

表是可以遍历的，如下所示：

```
7> ets:insert(tab,{racket,strict}).
true
8> ets:insert(tab,{ocaml,strict}).
true
9> ets:first(tab).
racket
10> ets:next(tab,racket).
haskell
```


因为这是 set 类型的 ETS 表，所以元素的顺序并非是以键来排序的；相反，顺序取决于表内部实现时产生的哈希值。在本例中，第一个键是 `racket`，接下来则是 `haskell`。但是如果使用 `first` 和 `next` 操作遍历的是有序集（ordered set）类型的表，则会依键的顺序进行遍历。使用 `match` 函数能够批量抽取信息：

```
11> ets:match(tab,{'$1','$0'}).
[[strict,ocaml],[ghci,haskell],[strict,racket]]
12> ets:match(tab,{'$1','_'}).
[[ocaml],[haskell],[racket]]
13> ets:match(tab,{'$1',strict}).
[[ocaml],[racket]]
```

第二个参数是一个符号（symbolic）元组，用于匹配 ETS 中的元组。结果是一个列表的列表，其中每个列表元素对应匹配命名变量 `'$0'` 的值，按升序排列；这些变量匹配元组中的任意值。通配值 `'_'` 也匹配任意值，但是它的参数不会在结果中出现。

我们来实现一个 `hlr` 模块，其中的代码能够使用 ETS 表把电话号码与 `pid` 关联在一起。更准确地说，这里的电话号码指的是 MSISDN（mobile subscriber integrated services digital network）号码。当电话附入（attach）网络时，我们创建电话号码与 `pid` 的对应，而当电话离开（detach）网络时我们将其删除。还允许用户查询 `pid` 对应哪个电话号码，反过来通过电话号码查询对应哪个 `pid`。建议认真阅读下面的代码，因为后面的章节我们将会把它作为另一个更大的例子的一部分：

```
-module(hlr).
-export([new/0, attach/1, detach/0, lookup_id/1, lookup_ms/1]).
```

```
new() ->
    ets:new(msisdn2pid, [public, named_table]),
    ets:new(pid2msisdn, [public, named_table]),
    ok.
```

```
attach(Ms) ->
    ets:insert(msisdn2pid, {Ms, self()}),
    ets:insert(pid2msisdn, {self(), Ms}).
```

```
detach() ->
    case ets:lookup(pid2msisdn, self()) of
    [{Pid, Ms}] ->
        ets:delete(pid2msisdn, Pid),
        ets:delete(msisdn2pid, Ms);
    [] ->
        ok
```

```
end.
```

```
lookup_id(Ms) ->
```

```
case ets:lookup(msisdn2pid, Ms) of
```

```
[] -> {error, invalid};
```

```
[[Ms, Pid]] -> {ok, Pid}
```

```
end.
```

48

```
lookup_ms(Pid) ->
```

```
case ets:lookup(pid2msisdn, Pid) of
```

```
[] -> {error, invalid};
```

```
[[Pid, Ms]] -> {ok, Ms}
```

```
end.
```

在下面的测试过程中，shell 进程把自身以号码 12345 附入了网络。我们分别使用号码和 pid 进行查找，然后离开了网络。当阅读代码时，注意我们使用的是命名的公开表，这意味着任何进程只要知道表名都能读取和写入：

```
2> hlr:new().
```

```
ok
```

```
3> hlr:attach(12345).
```

```
true
```

```
4> hlr:lookup_ms(self()).
```

```
{ok, 12345}
```

```
5> hlr:lookup_id(12345).
```

```
{ok, <0.32.0>}
```

```
6> hlr:detach().
```

```
true
```

```
7> hlr:lookup_id(12345).
```

```
{error, invalid}
```

分布式 Erlang

到目前为止，我们见到的例子都是在单台虚拟机——也称之为节点 (node)——上执行的。Erlang 内置了相关语意允许程序跨多节点执行：多个进程可以透明地在其他节点上分裂出进程，并相互通过消息传递来通信。分布式的各个节点可以都位于同一台物理（或虚拟）主机上，也可以分布在不同的主机上。

这一编程模型是为支持可伸缩和容错系统设计的，这些系统应运行在有防火墙保护的可信网络内。可见，在不加修改开箱即用的情况下，Erlang 的分布式不适用于如因特网、共享的云计算实例这类存在风险的环境下，在这种环境下进行跨系统的相互操作不属于 Erlang 分布式的设计初衷。毕竟不同的系统环境对安全有不同的要求，很难设计一种方

案能普遍适用。不过你可以通过提供自己的安全层和认证机制，或者通过修改 Erlang 的网络及安全库模块来容易地（也可能不那么容易）解决你对于安全方面的各种需求。

命名与通信

一个 Erlang 节点要想成为某个分布式 Erlang 系统的一部分，它必须拥有一个名字。通过 `erl -sname node` 方式启动的 Erlang 系统，使用的是短命名模式，这时当前节点将使用主机名（hostname）作为在本地网络的标识。与此相对的，启动节点时使用 `-name` 标志则是使用长命名模式，将使用一个全限定的（fully qualified）域名或 IP 地址。在一个分布式系统内，所有节点都必须使用同种命名模式，例如全是短命名，或者全是长命名。

49

位于分布式节点上的进程，其标识方式与在本地节点上时完全一样，都是使用 `pid`。这样的设计使得我们可以使用 `Pid!Msg` 给集群内任意节点上的进程发送消息。另外，为进程注册别名后，该别名会在所有主机上都可见，因此可以使用 `{bar, 'foo@myhost'}!Msg` 来向节点 'foo@myhost' 上的名为 `bar` 的进程发送消息。请注意该节点标识符的构成：由 `foo`（节点名）和 `myhost`（节点 `foo` 所在主机在本地网络中的名字）构成。

使用 `link(Pid)`、`spawn(Node, Mod, Fun, Args)`、`spawn_link` 函数时，你可以在系统内的任意节点上分裂、链接进程，并非仅限于本地节点。如果调用成功，`link` 将会返回原子 `true`，而 `spawn` 则返回进程在远程主机上的 `pid`。



代码本身不会自动被部署到远程节点！如果你在远程节点上分裂了一个进程，你有职责确保该远程节点所在的主机上已经正确地安放好编译好的代码，并位于相应的代码搜索路径之中。

节点间的连接与可见性

为了能够相互通信，Erlang 的节点间必须共享一个私密的 `cookie` 值。默认情况下，每个节点自己会生成一个随机的 `cookie` 值，除非你的主（home）目录下的 `.erlang.cookie` 文件里已经设定了一个值。当你首次启动分布式 Erlang 节点时会检查该文件，如果不存在则自动创建，并填入一个字符序列。这一行为可以被改变，只要启动时带上 `-setcookie Cookie` 标志就行，这里的 `Cookie` 指的就是具体的 `cookie` 值。除此方法外，还可以在程序中调用 `erlang:set_cookie(Node, Cookie)` 来进行设定。

在 Erlang 分布式系统里，默认情况下，如果一组节点共享相同的 `cookie` 值，它们中的任何一个节点便都能够知道其他所有节点的存在，并且可以相互交互。如果在启动节点时带上 `-hidden` 标志，这样该节点便不会自动与任何节点连接了，你可以根据需要手动

操作设定连接。使用 `net_kernel` 模块可以对此进行细粒度的控制，还能控制互联时的其他方面。隐藏式节点有多种用途，包括用于运维和作为桥（bridge）连接不同的节点集群。

在不同节点上的两个进程间进行消息投递时，消息顺序是有保障的，与在单节点内进行同样的通信相比，分布式系统下远程节点可能下线。为了应对这种问题，一种通用的做法是监视远程节点的存活性。这和本章前面“监视器”小节描述的监视本地进程不同，下面给出了一个例子：

50

```
monitor_node(Node, true),
{serve, Node} ! {self(), Msg},
receive
    {ok, Resp} ->
        monitor_node(Node, false),
        <handle process response>; % 伪代码：处理进程响应
    {nodedown, Node} ->
        <handle lack of response> % 伪代码：当收不到进程响应时
end.
```

在上述片段里，我们向 `Node` 指代的节点上的 `serve` 进程发送了一条消息（有可能代表一次远程过程调用的含义）。在发送请求前，我们对 `Node` 节点开始了监视，这样一来，如果该节点下线，我们将收到一条 `{nodedown, Node}` 消息，于是我们就能够处理无响应的情况了。一旦响应（`Resp`）成功收到，代码立刻关闭监视，然后才处理响应本身。你还可以使用 `monitor_node/2,3` 内置函数来监视远程节点，获得其健康状态的通知。

为了测试分布式通信，启动两个分布式 Erlang 节点，它们名字不同，但 `cookie` 值相同：

```
erl -sname foo -setcookie abc
erl -sname bar -setcookie abc
```

在下面的命令序列里，命令 1 `ping` 了远程节点，这同时也创建了与远程节点的连接。命令 2 使用 `nodes()` 内置函数查出了所有相连的节点，并绑定到了变量 `Node` 上。命令 4 在远程节点上分裂出一个新进程，该进程把其 `pid` 值发给了本地节点上的 `shell` 进程。我们在命令 5 里接收到了这一 `pid` 值，然后在命令 6 里使用 `node/1` 内置函数获取了对应的节点信息。命令 7 则在远程节点上分裂出一个新进程，该进程把该节点的标识符发回给当前节点。注意，节点名字都是原子，因为其中含有特殊字符，所以前后有单引号：

```
$ erl -sname bar -setcookie abc
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V7.2 (abort with ^G)
(bar@macbook-pro-2)1> net_adm:ping('foo@macbook-pro-2').
```



```

pong
(bar@macbook-pro-2)2> [Node] = nodes().
['foo@macbook-pro-2']
(bar@macbook-pro-2)3> Shell = self().
<0.38.0>
(bar@macbook-pro-2)4> spawn(Node, fun() -> Shell ! self() end).
<5985.46.0>
(bar@macbook-pro-2)5> receive Pid -> Pid end.
<5985.46.0>
(bar@macbook-pro-2)6> node(Pid).
'foo@macbook-pro-2'
(bar@macbook-pro-2)7> spawn(Node, fun() -> Shell ! node() end).
<5985.47.0>

(bar@macbook-pro-2)8> flush().
Shell got 'foo@macbook-pro-2'
ok

```

51

总结

本章我们对 Erlang 的基本知识进行了回顾，我相信这对于大家理解本书后续章节的内容很重要。并发模型、错误处理语意、分布式处理，这一切不仅使 Erlang 成为强大的工具，而且构成了 OTP 框架的基础。而具备能在运行时进行模块升级的能力无疑更是锦上添花。继续阅读后续章节前，我想强调，你对 Erlang 内部工作机制了解得越清楚，越有助于你学习 OTP 设计原则和本书介绍的其他知识，你的收获甚至会超出这些见解。我们提供了许多纯 Erlang 编写的例子，它们应该足以帮助你理解 OTP 的基本原理。多想多试，如果你遇到问题卡住了，建议你阅读 *Erlang Programming* 一书，这是一本书由 O'Reilly 出版社出版的书，合著者中的一人也是本书的作者。本书可以视为 *Erlang Programming* 的续篇，本书对源于该书中的许多例子做了展开探讨。其他同样有帮助的书包括 *Introducing Erlang*，也是 O'Reilly 出版的，作者是 Simon ST. Laurent；*Learn Your Some Erlang for Great Good* 一书由 No Starch Press 出版，作者是 Fred Hébert，这本书有免费在线版；以及 *Programming Erlang*，作者是 Erlang 的联合发明人 Joe Armstrong，由 The Pragmatic Bookshelf 出版。

接下来是什么

在接下来的章节里，我们将介绍进程设计模式和 OTP 的行为模式。我们首先提供了一个 Erlang 版客户端-服务器应用的例子，然后将其划分为通用的(generic)和专用的(specific)

两部分。通用部分可以打包为库模块，在不同的客户端 - 服务器应用间重用；专用的部分则与具体项目有关，需要针对不同需求去实现。在第 4 章，我们会改写这些代码，使用基于 OTP 的通用服务器行为（generic server behavior）来实现，这样就介绍了 Erlang 式系统的第一个构件。后续章节还会陆续介绍其他行为模式（behavior），随着这一过程的推进，你将对 Erlang 是如何架构的、各部分是如何相互衔接的有一个清晰的认识。

行为模式

在学习如何创建进程监督树结构以及如何为并发模型建立合适的架构之前，需要先花一些时间理解清楚行为模式（behavior）背后的那些设计原则。我们不会直接跃入函数和回调接口的世界开始介绍，而是侧重于解释各种幕后原理，确保你能够理解各种 OTP 行为模式的好处和优势，从而高效地使用它们。说了这么多，所谓的 OTP 行为模式到底是什么呢？

在 Erlang 中，许多不同的进程看起来解决的是不同的问题，但却遵循着某些相似的设计模式。其中特别常用的一些模式被抽象出来，封装为一组通用化的库模块，被称为 OTP 行为模式（behavior）。当读到 behavior 一词时，你可以认为它指的是某种形式化的进程设计模式。

严格来说，设计模式中的各种概念原本是用于面向对象编程领域的，不曾用于 Erlang 之上，但 OTP 隐藏并抽象了各种疑难问题以及复杂情况的处理逻辑，针对并发进程环境为我们提供了一套强大、可重用的解决方案。这样一来，借助这套可靠、仔细测试过、通用并且可重用的代码库，我们在项目中就不再需要重新发明轮子，使可维护性得以最大化。这些行为模式用“设计模式的话来说”就是实现了一套并发模型库。

进程的骨架

如果你试图比较这样两种 Erlang 进程，一种是管理键值存储的进程，另一种是管理复杂 GUI 系统中窗体的进程，那么乍一看二者的功能可谓截然不同，几乎没有多少相似点。但实际上这两种进程的生命周期却是相同的。它们都会：

- 分裂并初始化
- 反复地接收消息、处理消息并发送回应
- 终止（正常或非正常）

任何进程，不管用于何种目的，都必须先被分裂出来。一旦分裂了，它们会进行状态初始化。状态的具体内容则取决于该进程的用途。用于窗体管理的进程，则会绘制窗体并显示内容。用于键值存储的进程，则会创建空的表，然后把备份文件的——或者分布式集群中的其他节点上的其他表的——数据读取到其中。

一旦进程初始化完毕，就可以接收各种事件了。这些事件多种多样，以窗体管理进程为例，可以是窗体输入框中的键盘按键事件、按钮单击事件、菜单项选择事件等，还可以是移动窗体或移动其中元素时产生的拖动与释放事件。这些事件都可以用 Erlang 中的消息表示。当收到消息后，进程就会处理该请求——即根据消息的内容执行操作并更新进程自身的内部状态。于是当你按键时内容才会显示到屏幕上，当你单击按钮、选择菜单项时窗体才会相应更新，当你拖曳时目标才会在屏幕上按你的意图移动。键值存储进程的工作方式也与此类似。通过向键值存储进程发送异步消息，可以在表中插入或删除元素，而发送同步消息（接收方收到这种消息后需要发送回应）则可以查询某些元素并把它们的值返回给客户端。

最终，进程会终止。原因可能是因为用户单击了菜单中的关闭项，或者单击了窗体上的关闭按钮。当出现这些情况时，窗体管理器将会释放为该窗口分配的资源，然后窗体被隐藏或者关闭。一旦清理过程执行完毕，进程就不再有需要执行的代码了，因此它就正常终止了。而对于键值存储进程的终止，则很可能是因为收到了一条 stop 消息导致的，这条消息使得进程在终止前先把表中的内容备份到另外的节点上，或者存储到持久化媒介上。

进程也可能会非正常终止，原因可能是出现异常，或者相链接的其他进程发来了 exit 信号。在这个过程中，进程也可能可以通过设定 trap_exit 标志或者使用 try-catch 表达式，使得进程可以避免因为 exit 信号或异常而直接地“非正常”终止。这样一来，进程可以立刻或者稍后自行调用某些命令转变成以“正常”方式终止——这些命令也正是一般情况下进程正常终止时所调用的那些。我们之所以说“可能可以”，是因为毕竟计算机电源线可能会被拔掉，硬盘可能会出故障，管理员可能会被网线绊倒，进程可能收到原因被设置为 kill 的 exit 信号——这会导致进程被无条件终止，你不会有机会进行额外的控制。

图 3-1 展示了一个典型的进程流程图，它反映了进程的整体生命周期。

55

正如我们已经描述过的，即使各类进程需要完成的任务并不相同，但它们都会以与此相似的方式执行，遵循一定的模式。而遵循这些模式的结果就是，这些进程可以共用一些相似的基础代码。Erlang 进程循环的典型模式为，启动、处理事件，最后退出，类似这样：

```
start(Args) ->                                % 启动 server 进程
    spawn(server, init, [Args]).

init(Args) ->                                  % 初始化内部的进程状态
```



```

State = initialize_state(Args),
loop(State).

loop(State) ->                                % 接收消息并处理
    receive
        {handle, Msg} ->
            NewState = handle(Msg, State),
            loop(NewState);
    stop ->
        terminate(State)                       % 停止进程
    end.

terminate(State) ->                            % 在停止前做清理
    clean_up(State).

```

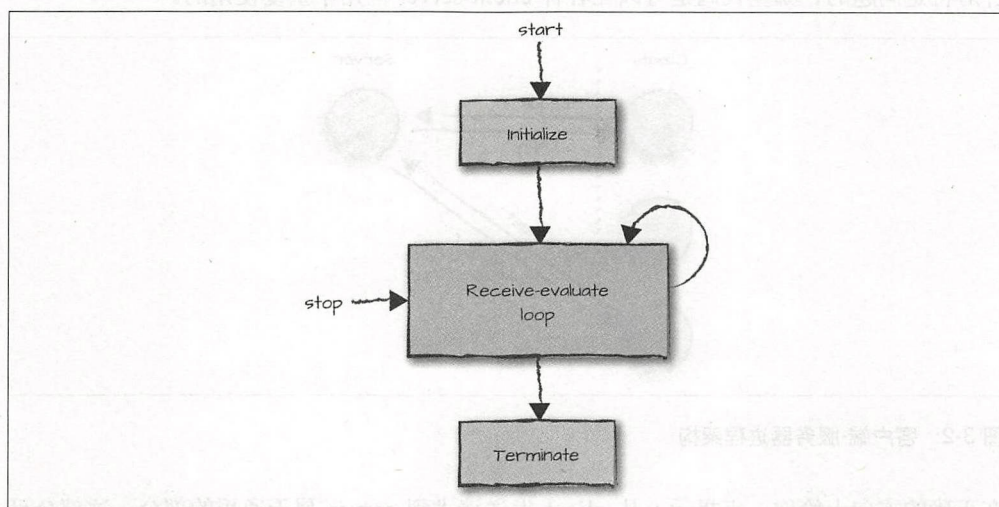


图 3-1: 进程的骨架

这样一种典型的客户端 - 服务器行为模式，在 Erlang 中被称为 **client-server behavior**。server 启动后，一旦收到消息，其 `handle/2` 函数就会被调用；然后在该函数中，会发送回应（如果有必要），然后更新状态；最后继续循环以准备处理下一条到来的消息。一旦收到 `stop` 消息，进程会清理自身资源然后终止。

虽然我们说这是 Erlang 中一种典型的 **client-server behavior**，但实际上这一模式是其他一切模式的基础。它无处不在，即使那些根本不使用 OTP 行为模式库的代码里也倾向于使用与其相同的函数名。这使得不管是谁都能在阅读代码时清楚明白地知道，进程状态是在 `init/1` 中初始化的，消息是在 `loop/1` 中接收的，然后每一条消息则是通过调用 `handle/2` 函数处理的，`terminate/1` 函数则负责完成各种资源的清理。这样的代码，以

◀ 56

后当有人试图维护时很快就能明白进程的基本行为，而无须了解任何通信协议、底层架构或者进程结构。

设计模式

现在,让我们通过更多例子深入地剖析 Erlang 中 client-server 架构是如何实现的。实际上,在 Erlang 中,client 和 server 都是进程,它们的请求和响应是靠发送消息进行的。看图 3-2,回想一些你曾经做过的或者读过的 client-server 架构,优先找看上去相似性不多的那些(就像我们举的键值存储和窗体管理的例子那样)。把注意力集中在应用中那些与 Erlang 相关的构造和模式的代码上,尝试列出不同实现中的相似之处和不同点。问问自己,这些代码中哪些部分是通用的(generic),哪些是专用(specific)的,哪些代码是针对特定问题的,哪些代码是可以在各种 client-server 应用中重复使用的。

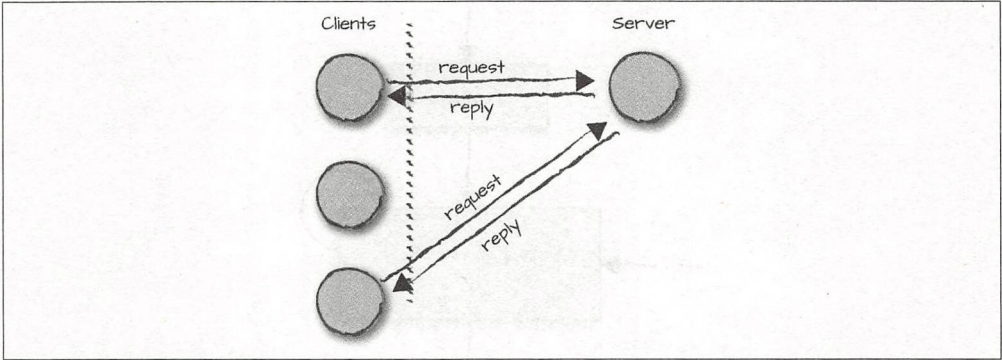


图 3-2: 客户端-服务器进程架构

在正确的方向上给你一点提示:从 client 发送请求到 server 属于通用的部分。这部分可以统一为使用某种方案,能够适用于任何 client-server 架构,并且无所谓 server 到底是做什么的。然而,具体到请求消息中的内容,则属于专用部分。

我们从分裂出 server 开始。创建一个进程,然后调用 `init/1` 函数这一点是通用的。而在调用时具体传入的参数是什么,以及 `init/1` 函数中初始化进程状态并返回循环数据的具体过程则都是专用的。循环数据扮演了变量的角色,其在各次调用之间负责存储进程数据。

在各次调用之间存储循环数据这一点,进程们都普遍需要,但是具体到存储何种循环数据则各不相同。不仅会由于各种不同进程所要完成的任务不同而不同,甚至对于完成相同种类任务的不同进程实例来说,也会有差异。

发送请求给 server，然后 server 做出回应，属于通用部分。而具体到发送给 server 的请求的类型、内容，以及 server 处理的过程和发送回 client 的响应则属于专用的。不过虽然响应的内容是专用的，发送响应给 client 进程这一行为本身却是通用的。

server 应该是可以被停止的。其中发送 stop 消息、处理异常和发送 EXIT 信号部分是通用的，而终止前调用哪些函数清理状态则是专用的。

表 3-1 总结了 client-server 架构中哪些部分是通用的，哪些部分是专用的。

表 3-1: client-server 中的通用代码与专用代码

通用	专用
分裂 server 进程	初始化 server 状态
存储循环时的数据	循环时的数据的具体内容
发送请求给 server	client 发送的请求内容
发送回应给 client	处理 client 请求的过程
接收 server 的回应	server 回应的内容
停止 server	终止前的资源清理过程

回调模块

OTP 行为模式的背后是这样一种理念——把代码分割为两类模块：一类负责实现通用的部分，我们称此类模块为 behavior 模块；另一类负责实现专用的部分，我们称此类模块为回调模块（参见图 3-3）。可以把通用的 behavior 模块形象化地理解为“驾驶员”。虽然它不知道回调模块具体做些什么，但它知道回调模块中导出了一组回调函数，它必须去调用这些函数，并且它知道这些函数的返回值的格式。而反过来，回调模块也并不清楚 behavior 模块具体做了什么；回调模块只是遵循约定，当被调用时返回符合格式的数据。

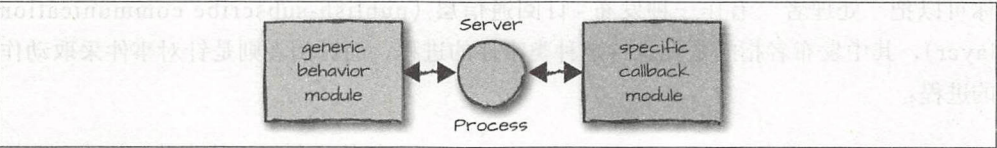


图 3-3: 回调（callback）模块

换个角度，我们也可以把这一切理解为 behavior 模块和回调模块之间的一种“契约”。58 两者必须针对回调 API 中涉及的函数约定好一组名称、类型及相应的返回值。

behavior 模块中包含的是那些通用的功能，可以在各个实现间反复使用。OTP 以库模块的方式提供了许多种 behavior。而回调模块则由应用开发人员负责实现。其中包含了所有特定于所要实现的进程的代码。

OTP 提供了 5 种 behavior，涵盖了所有情况中的绝大部分。它们是：

`gen_server`，通用 Server

用于建立 client-server 行为模型。

`gen_fsm`，通用有限状态机 (Finite State Machine)

用于有限状态机编程。

`gen_event`，通用事件处理 / 管理器

用于编写事件处理器。

`supervisor`，监督者

用于构建具备容错能力的监督树 (supervision tree)。

`application`，应用

用于整合资源与功能。

`gen_server` 是最常用的 behavior。对于 client-server 架构，可以用 `gen_server` 建立其进程模型，我们已经讨论过的键值存储和窗体管理器都适用于此方式。

`gen_fsm` behavior 为涉及 FSM (有限状态机) 的工作提供了全套所需的通用构造。开发人员通常使用 FSM 来实现自动化控制系统、协议栈以及决策系统。FSM 中的代码可以手工实现，或者由其他程序生成。

`gen_event` 用于事件驱动编程，其中的事件是以消息方式接收的，然后会触发相应的一个或多个动作，这些动作被称为“处理者”。典型的“处理者”完成的功能诸如日志记录、指标收集、告警等。

59 > 你可以把“处理者”看作一种发布 - 订阅通信层 (publish-subscribe communication layer)，其中发布者指的是发送特定种类事件的进程，而订阅者则是针对事件采取动作的进程。

`supervisor` (监督者) 属于一类特别的 behavior，它们的任务仅仅是启动、停止和监视一些“孩子进程”，这些“孩子进程”一般来说是一些“工作者”进程，但也有可能是其他 `supervisor` 进程。允许一个监督者监视另一个监督者，使得我们能够构造出一种称为监督树的进程结构。在后面的章节中我们会详细介绍监督树。监督者重启孩子进程时，使用的配置参数由回调函数提供。

监督树打包为一个整体后，置于一种我们称为 `application` 的 behavior 中。`application` 负责启动顶层的监督者，把相互依赖的进程封装为一个整体，运行于 Erlang 节点中。

`gen_server`、`gen_fsm`、`gen_event` 这些都可以被说成是“工作者”：负责大部分计算任务的进程。它们被 `supervisor` 和 `application behavior` 共同持有。如果你需要的 `behavior` 在标准库里并未提供，你可以自己实现它们，但需要遵循一组特别的规则以及指令，在第 10 章有详细介绍。我们称它们为“特殊进程”（special process）。

此刻的你心里或许很疑惑：在软件里增加这样一层复杂性目的何在？原因很多。使用 `behavior`，我们一方面把编程风格标准化了，另一方面减少了开发大型软件时所需要编写的代码量。通过把所有这些通用的设计模式封装在库模块中，我们可以重用代码，减少开发时的负累。我们使用的这些 `behavior` 从 20 世纪 90 年代中期就在生产环境中使用至今，是坚固、良好测试过的。它们内部处理了并发时的所有棘手细节，无须程序员再操心。因此，建立在它们之上的系统故障^{注 1} 更少，同时拥有容错的基础。另外，这些 `behavior` 还内置了一些额外的功能，如日志记录、追踪、统计等，并可以进行功能扩展，进而使得所有使用这些 `behavior` 的进程都受到影响。

使用这些 `behavior` 还有另一个重要的优势，它们都提倡使用一种共用的编程风格。不管是谁，只要阅读回调模块中的代码就会立刻明白，进程中的状态是在 `init` 函数中初始化的，而 `terminate` 函数中包含的则是当进程停止时执行的清理代码。它们会了解到通信协议是怎样的，出错时进程是如何重启的，监督树是如何打包的。特别是当编程规模变得巨大时，这种编程风格使得所有读代码的人可以基于自己对通用部分代码已知的知识，识别并排除通用部分，把精力集中在与项目紧密相关的特定部分。这样一种共用的编程风格还带来了一系列“组件式术语”，使得各个分散的团队能使用一套标准化的词汇进行沟通。说到底，时间大部分其实都是花在阅读和维护代码上，而不是编写。当你打造一套高度复杂、不可失效的系统时，让代码易于理解绝对是有必要的。

60

那么，介绍了如此之多的优点后，缺点是什么？学会正确、熟练地使用这些 `behavior` 是有一定困难的。学会如何恰当地使用 OTP 设计原则创建系统需要花费一些时间，但随着文档的改良，训练课程、书籍以及工具越来越丰富。你正在阅读的这本 OTP 主题的书不正也说明了这一点吗？

`behavior` 的存在使得调用链又略微增加了几层，并使得收发消息时的数据量有些许增加。虽然这可能会影响性能和内存用量，但在绝大多数情况下影响可以说微不足道，特别是考虑到这些 `behavior` 对代码整体质量和功能的改善后。写出更快但是错误更多的代码到底又有什么意义呢？这些内存用量的小小增加以及性能的些许降低实质上都是为了可读性和容错所付出的小小成本。根本原则永远是——从用 `behavior` 开始，当瓶颈出现后再

注 1 完美无错（Bug）的系统只存在于官僚主义者的梦里。当使用 Erlang/OTP 时，我们除了需要投入精力保障系统的正确性之外，也应当投入等量的精力在系统的错误恢复方面，因为故障是无情的，它们会在生产环境中出现。

优化。你将会发现，由于 behavior 内部代码不够高效而导致需要你去优化这种情况基本不会出现。

抽取出通用的行为模式

介绍完 behavior，现在让我们一起看一个熟悉的 client-server 的例子，它是以纯 Erlang 编写的，没有使用 behavior。我们使用的这个“频率分配服务器”（frequency server）的例子原本是 *Erlang Programming* 一书中的，实现在名为 frequency 的模块里。如果你因为没读过那本书而不熟悉这一例子，请别担心，我们会随着内容的展开而逐渐介绍。在这个例子中，server 部分为手机提供频率分配服务。当拨打手机时，为了相互间能够建立通信连接，需要为此次会话分配频率。手机作为 client 会一直持有此频率，直到会话结束，然后回收频率供其他订户重用（参见图 3-4）。

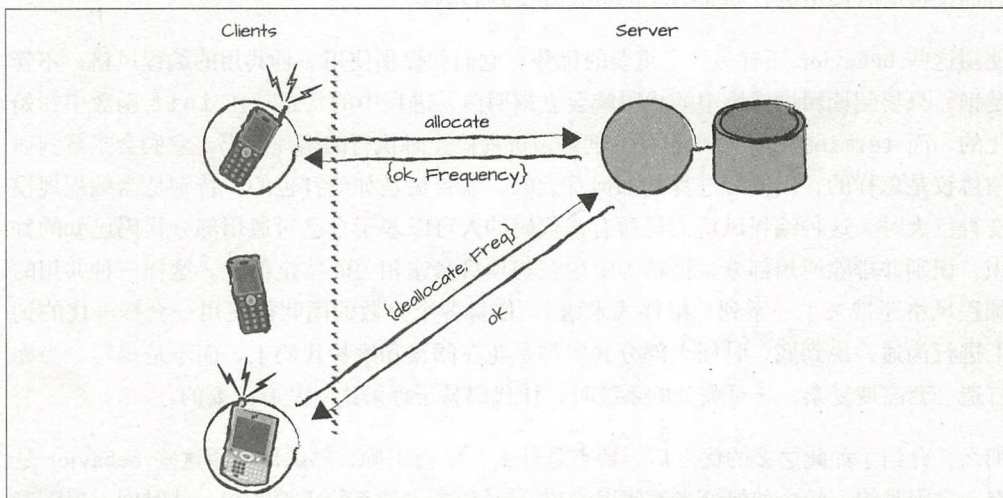


图 3-4：频率服务器

考虑到这是本书中第一个主要的例子，因此我们将会比一般例子介绍得更细致一些。在后续章节中，我们的速度不会这么慢，所以如果你的 Erlang 知识有点生疏了，可以借这个机会补一补。我们会列出频率分配服务器例子的代码，然后找出其中的通用部分，将其抽取出来放到单独的库模块中去。最终产出的将是两个模块：一个包含的是通用的、可重用的代码，另一个包含的则是与频率分配服务逻辑本身紧密相关的专用的代码。

61 众多 client 以及唯一的一个 server 实质上都是一些 Erlang 进程，它们相互之间的信息交换都是借助传递消息达成的，而传递消息的过程又隐藏在一些函数接口中。client 用到的函数接口包括 `allocate/0` 和 `deallocate/1`：


```
allocate() -> {ok, Frequency} | {error, no_frequency}
deallocate(Frequency) -> ok
```

allocate/0 函数返回 {ok, Frequency} 作为结果,当然前提是至少有一个可用频率。否则若所有频率都在使用中,则返回 {error, no_frequency}。当 client 完成通话后,它可以通过调用 deallocate/1 函数释放频率,调用时需要传入 Frequency 作为参数。

我们调用 start/0 来启动 server,然后在之后某个时刻调用 stop/0 来终止它:

```
start()-> true
stop() -> ok
```

server 会以 frequency 作为别名被静态注册,因此向其传递消息时直接用别名即可,无须记忆 pid。

从 shell 里试验 frequency 模块看起来大概是这样——首先启动 server,然后分配完全部共 6 个频率,于是在第 7 次分配时失败。这时候我们释放回频率 11,因此又可以再成功分配一次。最后通过停止 server 来终止试验:

```
1> frequency:start().
true
2> frequency:allocate(), frequency:allocate(), frequency:allocate(),
   frequency:allocate(),frequency:allocate(), frequency:allocate().
{ok,15}
3> frequency:allocate().
{error,no_frequency}
4> frequency:deallocate(11)..
ok
5> frequency:allocate().
{ok,11}
6> frequency:stop().
ok
```

62

如果你渴望对代码进行更深入的探索,可以自己去下载和运行由本书代码仓库中提供的模块源码。接下来,我们把代码过一遍,详解其中做了什么,然后把通用部分和专用部分分开。

启动 server

让我们以创建和初始化 server 的函数为起点出发吧。start/0 函数会分裂出一个进程,该进程一方面会调用 frequency:init/0 函数,另一方面会为自身注册上 frequency 别名。init 函数主要负责初始化进程状态,该状态实质上是一个元组,包含可用频率列表(为

为了方便演示我们把值硬编码在了 `get_frequencies/0` 函数里), 以及已分配频率列表 (一个空列表)。我们把这个元组绑定到 `Frequencies` 变量上, 并且在后面的例子中把它统称为进程状态 (process state) 或者循环数据 (loop data)。随着循环的每一次迭代, 进程状态发生改变, 频率在“可用”和“已分配”两个列表间移动。

注意两点, 第一, 我们导出了 `init/0` 函数, 因为我们需要把它作为参数传递给 `spawn BIF`; 第二, 我们为 `server` 进程注册了别名, 并且使用的名称正是模块名本身。对于第二点, 并不是强制要求, 但是这么做被认为是一种好的 Erlang 编程实践, 因为这有助于对线上系统进行调试和检修。

```
-module(frequency).  
-export([start/0, stop/0, allocate/0, deallocate/1]).  
-export([init/0]).  
  
start() -> register(frequency, spawn(frequency, init, [])).  
  
init() ->  
    Frequencies = {get_frequencies(), []},  
    loop(Frequencies).  
  
get_frequencies() -> [10,11,12,13,14,15].
```

看看前面这些代码, 试着指出其中通用的部分。这些部分不会因为更改了 `client-server` 的实现方式而发生变化, 你能找出来吗?

从 `export` 指令处开始看, 你应当意识到, `start` 和 `stop` 功能总是必需的——无论 `server` 具体做的是什。因此, 我们可以认为这些函数是通用的。基于同样的道理, 那些分裂进程、注册别名、调用初始化函数 (初始化进程状态) 的部分也都是通用的。进程状态将会被绑定到某个变量上, 并且在进程循环过程中传递。但是, 虽然我们说一些函数和 BIF 是通用的, 但是这些函数内部的过程, 以及传递给这些函数的参数却不是通用的。根据具体的 `client-server` 实现情况, 有不同的变化。在下面的代码里, 我们加粗了所有通用的部分:

```
-module(frequency).  
-export([start/0, stop/0, allocate/0, deallocate/1]).  
-export([init/0]).  
  
start() ->  
    register(frequency, spawn(frequency, init, [])).  
  
init() ->  
    Frequencies = {get_frequencies(), []},
```



```
Loop(Frequencies).
```

```
get_frequencies() -> [10,11,12,13,14,15].
```

了解了通用部分，我们接下来看看专用部分，也就是前面代码中那些没有加粗的部分。首先跃入眼帘的，是模块名 `frequency`，它是特定的。显然不同的 `server` 实现会有不同的模块名。为 `client` 提供的 `allocate/0` 和 `deallocate/1` 函数也是此 `client-server` 应用特有的，你不太可能会在窗体管理器或者键值存储的应用中看到它们（即使真的看到了相同的名字，那些函数实质上所做的是肯定是完全不同的）。尽管启动 `server`、分裂 `server` 进程、注册别名等这些是通用的，但是具体注册成什么别名，以及是在哪个模块里含有 `init` 函数等这些都是特定的。

传递给 `init` 函数的参数也是特定的。在我们的例子里，没有传递任何参数（因此其元数为 0），但其他 `client-server` 实现中不一定是这样的。`init/0` 函数中的表达式是用于初始化进程状态的。不同的实现，初始化状态时所做的事情不同。初始化窗体设定并显示窗体；创建空的键值存储、上传持久化备份；或者是像本例中那样，生成一个元组，其中包含了可用频率的列表等。

当进程状态已经初始化完成后，会把它绑定到某个变量上。存储进程状态这一行为可以认为是通用的，但是状态中的内容却是特定的。在下面的代码里，我们加粗了变量 `Frequency`，认为它是特定的。这意味着变量的内容虽然是特定的，但是把它传递给进程循环这一机制却是通用的。最后，`init/0` 里对 `get_frequencies/0` 的调用也是特定的。在真实世界的实现中，我们可能会从配置文件或者持久化的数据库，或者通过查询基站来读取频率列表。本例中主要是为了便于讨论，我们采取了直接硬编码到模块里的偷懒做法。

64

让我们加粗特定的代码：

```
-module(frequency).  
-export([start/0, stop/0, allocate/0, deallocate/1]).  
-export([init/0]).
```

```
start() ->  
    register(frequency, spawn(frequency, init, [])).
```

```
init() ->  
    Frequencies = {get_frequencies(), []},  
    loop(Frequencies).
```

```
get_frequencies() -> [10,11,12,13,14,15].
```

从我们强调的内容里，你是否看出了一些模式和思路？接下来，我们继续使用上述方法

对余下的模块进行分析，从为 client 提供的那些函数开始。

client 函数

我们把 server 模块导出提供给 client 进程调用的那些函数称为 *client API*，通过使用它们，client 进程可以控制和访问由 server 进程提供的服务。把消息传递和协议封装到函数内再对外提供是比较好的方案，有利于改善可读性和可维护性。我们例子中的这些 client 函数所做的正是这样的事。事实上，我们此处所做的甚至更进一步，把发送请求以及接收回应的操作也分别封装成了 `call/1` 和 `reply/2` 函数。如果不这么封装好，那么任何需要收发消息的地方都得写上一样的代码：

```
stop()          -> call(stop).
allocate()      -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).
```

```
call(Message) ->
    frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply end.
```

```
reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

`stop/0` 函数发送原子 `stop` 给 server。而 server 一旦在“接收 - 求值”（*receive-evaluate*）循环中接收到 `stop`，就会采取相应的动作。考虑到可读性和可维护性，最好在实践中使用一些能够表明意图的清晰的词汇，话虽如此，我们就算用别的（例如原子 `foobar`）也未尝不可，重点并非在于原子的名字，而在于其含义是否反映了其在我们程序中的用意。在我们的例子中，`stop` 的目的就是让进程正常终止。我们会在之后看到处理 `stop` 时的具体过程。

65 client 函数 `allocate/0` 和 `deallocate/1` 的调用和执行都是发生在 client 进程内的。client 通过执行 `frequency` 模块中提供的 client 函数发送消息给 server。消息本身作为参数传递给 `call/1` 函数，并且绑定到 `Message` 变量。`Message` 紧接着被插入到一个结构为 `{request, Pid, Message}` 的元组中，其中的 `Pid` 是 client 进程的标识符（通过调用 `self()` BIF 可获得），当 server 发送结构为 `{reply, Reply}` 的回应时使用该标识符指向的进程作为目的地。我们把类似这样在 client 和 server 之间通信时所做的填充行为称为“协议”（参见图 3-5）。

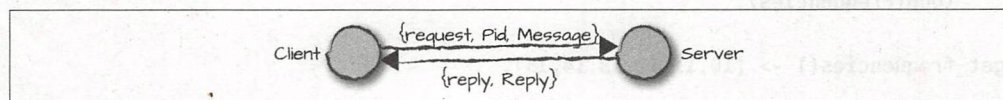


图 3-5：消息协议

server 接收请求并处理后，调用 `reply/2` 发送回应。调用时传入的第一个参数正是 client 请求中的那个 `pid`，而传入的第二个参数则是响应的内容本身。这条消息会在 `call/1` 函数中的 `receive` 分句处被模式匹配，然后取出其中的内容绑定到 `Reply` 上作为结果。这也就是该 `client` 函数最终返回的结果。图 3-6 展示了在电话和频率服务器之间进行消息交换的时序图（sequence diagram）。

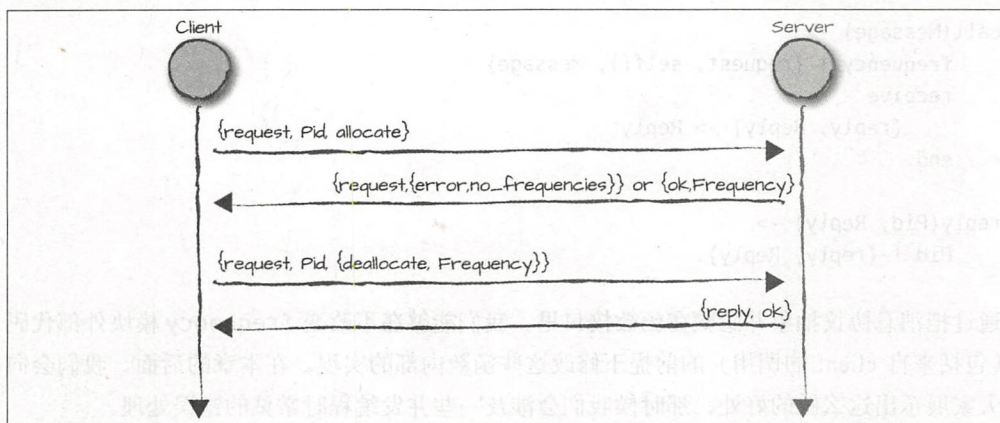


图 3-6: 频率服务器之前的消息

那么，代码中的哪些部分是通用的？哪些部分不会因为 client-server 的实现不同而改变？首先是 `stop/0` 函数，当我们想通知 server 终止时会使用这一函数。它可以被重用，因为这一需求是普遍性的。另外，每次想要发送消息时，我们会使用 `call/1` 函数。但是由于存在一点点问题，导致这一函数其实并不是完全通用的。仔细看看代码，试着指出问题所在：

```

stop()          -> call(stop).
allocate()      -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).
  
```

```

call(Message) ->
  frequency ! {request, self(), Message},
  receive
    {reply, Reply} -> Reply
  end.
  
```

```

reply(Pid, Reply) ->
  Pid ! {reply, Reply}.
  
```

我们把消息发给了一个名字注册为 `frequency` 的进程。这一名字是会随着实现不同而变化的。但除此之外，`call` 函数里的其他部分都是通用的。至于被 server 调用的 `reply/2`

函数，则是完全通用的。所以在 client 函数里，剩下的函数都是专用的，而且发往 server 的消息内容，以及 server 的名字等也是专用的。

```
stop()          -> call(stop).
allocate()      -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).
```

```
call(Message) ->
  frequency ! {request, self(), Message},
  receive
    {reply, Reply} -> Reply
  end.
```

```
reply(Pid, Reply) ->
  Pid ! {reply, Reply}.
```

通过把消息协议抽象并隐藏在函数接口里，我们能够在不改变 frequency 模块外部代码（包括来自 client 的调用）的前提下修改这些函数内部的实现。在本章的后面，我们会向大家展示出这么做的好处，那时候我们会涉及一些并发编程时常见的错误处理。

server 循环

server 进程在内部运行着“接收 - 求值” (receive-evaluate) 的循环。它们等待着来自 client 的请求，进行处理，返回结果，然后再次重复这一过程，等待下一条消息的到来。每一次迭代，它们会更新进程状态，并且会产生一些副作用：

```
loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
      {NewFrequencies, Reply} = allocate(Frequencies, Pid),
      reply(Pid, Reply),
      loop(NewFrequencies);
    {request, Pid, {deallocate, Freq}} ->
      NewFrequencies = deallocate(Frequencies, Freq),
      reply(Pid, ok),
      loop(NewFrequencies);
    {request, Pid, stop} ->
      reply(Pid, ok)
  end.
```

在频率服务器这个例子里，loop/1 函数接收 allocate、{deallocate, Frequency}、stop 三种命令。分配频率这一操作是通过 allocate/2 函数完成的，只要提供循环数据以及 client 的 pid，这一函数就会把频率从可用列表移到已分配列表。释放频率时调用的

deallocate/2 函数则做的是相反的操作——把频率从已分配列表移到可用列表。

调用这两个函数都会返回更新后的频率列表，这也正是进程的状态。此新状态被绑定到变量 NewFrequencies 并在尾递归调用 loop/1 时作为参数传入。在这两种情况（分配或释放频率）下，都会发送回应给 client。当分配频率时，变量 Reply 的内容要么是 {error, no_frequency}，要么是 {ok, Frequency}。当释放频率时，server 发回的则是原子 ok。

当停止 server 时，通过回应 ok 来确认我们已经接收到了消息，并且故意不去调用 loop/1 使得进程可以正常终止，与之相对的是非正常终止——那是由于运行时错误导致的。在这个例子里，没有什么需要清理的，因此我们除了确认 stop 消息外不做任何其他事。倘若此 server 完成的是键值存储方面的功能，那么我们则可能需要确保数据已经安全地备份到持久化媒介上后再终止。或者 server 完成的是窗体管理方面的功能，那么我们在终止前得关闭窗体并释放为窗体所分配的一些资源。

了解这些后，哪些功能你认为是通用的呢？

首先，“循环”这一行为本身是通用的。发送和接收消息时所用的协议是通用的，但是消息和回应自身却不是通用的。最后，收到 stop 消息后停止 server 这一行为是通用的。下面把代码中通用的部分加粗了：

```
loop(Frequencies) ->
  receive
  {request, Pid, allocate} ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    reply(Pid, Reply),
    loop(NewFrequencies);
  {request, Pid, {deallocate, Freq}} ->
    NewFrequencies = deallocate(Frequencies, Freq),
    reply(Pid, ok),
    loop(NewFrequencies);
  {request, Pid, stop} ->
    reply(Pid, ok)
end.
```

68

我们没有加粗 Frequencies 和 NewFrequencies 变量，它们是由于存储进程状态的。虽然存储进程状态这一行为是通用的，但是状态本身却是特定的。这指的是状态变量的类型和值是特定的，但是存储进程状态这一任务本身却是通用的。

既然通用的部分已经清楚了，那么剩下的部分，循环数据、client 消息、处理消息的过程，以及发送回去作为结果的响应等这些都是特定的：

```
loop(Frequencies) ->
```

```

receive
  {request, Pid, allocate} ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    reply(Pid, Reply),
    loop(NewFrequencies);
  {request, Pid, {deallocate, Freq}} ->
    NewFrequencies = deallocate(Frequencies, Freq),
    reply(Pid, ok),
    loop(NewFrequencies);
  {request, Pid, stop} ->
    reply(Pid, ok)
end.

```

例子里停止 server 后不再执行什么代码，如果有的话，那些代码也会被标记为特定的。而且那些代码通常会放在一个名为 `terminate` 的函数里，这一函数接受终止原因以及循环数据两个参数，负责处理和清理有关的所有事。

server 内部函数

在 server 模块内部有一些函数，它们才是实现频率分配和释放逻辑的地方，但这些函数却仅在模块内可见，外部无法访问，我们把它称为 server “内部” 的函数。调用 `allocate/1` 会返回一个元组，其中包含新的频率列表，以及发送回 client 的回应。如果没有可用的频率，第一个函数分句将会匹配，因为列表是空的。频率列表不会改变，`{error, no_frequency}` 被返回给 client。只要有一个或以上的频率，第二个函数分句就会匹配。

可用频率列表被分为头和尾两部分，头包含的是本次分配的频率，而尾则是剩下的可用频率列表（可能为空列表）。本次分配的频率会和 client 的 pid 一起被记录到已分配列表中，然后发送回应 `{ok, Freq}` 给 client。

69 当 `deallocate/2` 函数释放频率时，它会从已分配列表中找到对应频率，将其移动到可用频率列表里。看一看函数中的代码，试着指出哪些是通用的，哪些是专用的^{注1}：

```

allocate([], Allocated, _Pid) ->
  {[], Allocated}, {error, no_frequency}};
allocate([_Freq|Free, _], Allocated, Pid) ->
  {[Free, [{Freq, Pid}|Allocated]}, {ok, Freq}}.

deallocate([Free, Allocated], Freq) ->
  NewAllocated = lists:keydelete(Freq, 1, Allocated),
  {[Freq|Free], NewAllocated}.

```

注1 警告，这是一个陷阱问题。

这应该是一个很容易回答的问题，因为这些内部函数全都是特定于我们的频率分配服务器的。你什么时候在键值存储或者窗体管理服务里见过频率分配和释放服务的？

通用服务器

至此，我们已经把例子过了一遍，并且区分出了通用的和专用的代码，接下来我们进入到本章的核心部分，把这两类代码放到两个模块中并命名。如图 3-7 所示，我们可以把所有通用的代码放到 `server` 模块中，而将所有专用的代码放到 `frequency` 模块中。虽然做了这些改变，但是功能和接口依然和之前一样。调用新实现的 `frequency` 模块和之前在本章前面“抽取出通用的行为模式”一节介绍的完全一样。

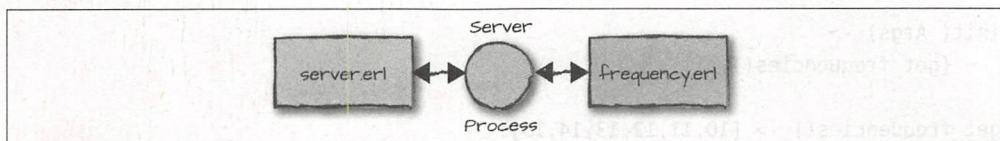


图 3-7: `frequency` 模块与 `server` 模块

`server` 模块主要负责控制、管理进程活动。当需要处理专用的功能时，它并不知道如何处理，只是通过回调函数把处理交给 `frequency` 模块。让我们从 `server` 模块中负责启动和初始化的通用代码开始：

```
-module(server).                                % server.erl
-export([start/2, stop/1, call/2]).
-export([init/2]).

start(Name, Args) ->
    register(Name, spawn(server, init, [Name, Args])).

init(Mod, Args) ->
    State = Mod:init(Args),
    loop(Mod, State).
```

分裂进程、注册、调用 `init` 函数都是通用的，其中涉及的注册进程所用的别名、回调模块的名称，以及传递给 `init` 函数的参数则都是专用的。我们把这些信息以参数方式传入 `server:start/2` 函数，然后使用。Name 参数既作为 `frequency` 进程的名称用到，同时也用作回调模块的名称。Args 参数则被传递给 `init` 函数，用于初始化进程状态。

我们把面向 client 的函数都保留在 `frequency` 模块里，对 `server` 模块的使用是包装在这些函数内部的。通过这种做法，我们隐藏了实现细节——包括那些使用 `server` 模块的过

程。就像先前例子里那样，我们通过使用 `frequency:start/0` 函数就能启动 `server`，因为里面暗含了对 `server:start/2` 的调用。新分裂的 `server`，通过 `Mod:init/1` 调用，调用了 `frequency` 模块中的 `init/1` 回调函数来初始化进程状态，其中创建了一个元组，元组中包含可用频率列表和已分配频率列表。`Mod` 绑定的是回调模块 `frequency`，而 `Args` 绑定的是 `[]`。包含了频率信息的元组则被绑定到了 `State` 变量，然后这一变量和 `Mod` 变量一起作为参数传递到了 `server` 模块的循环中：

```
-module(frequency).                                % frequency.erl
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/1, terminate/1, handle/2]).
```

```
start() -> server:start(frequency, []).
```

```
init(_Args) ->
    {get_frequencies(), []}.
```

```
get_frequencies() -> [10,11,12,13,14,15].
```

为了能够返回进程的初始状态，`init/1` 回调函数是必需的，返回的进程初始状态会存储下来在 `server` 的“接收 - 求值”循环中使用。在 `init/1` 回调函数中，我们并没有用到 `_Args` 参数的值。因为 `init/1` 是一个回调函数，我们必须遵从与该回调 API 对应的协议要求以及函数接口要求。就一般情况而言，`init/1` 必须接收一个参数，因为有些 `server` 实现可能会在启动时用到该数据。但在这个例子里并不需要，所以我们传递了一个空的列表，并且忽略了它。

让我们跳回到 `server` 模块。当 `client` 进程想要发送请求给 `server` 时，它通过调用 `server:call(frequency, Msg)` 完成。而 `server`，当需要发送回应时，则通过调用 `reply/2` 完成。从效果上来看，我们把所有消息传递的细节都隐藏到了函数接口后。

71 另一个通用的函数是 `server:stop/1`。把这个函数与 `call/2` 区分开是因为我们想从含义上做一些变化，如果直接使用 `server:call(frequency, {stop, self()})` 这种形式会让开发人员感觉和发送一般的 `server` 控制消息并无不同，但我们希望让他们意识到这一调用的特殊性。所以，我们要求他们必须调用 `stop` 函数来实现终止（而无法调用一般的 `call/2` 函数来实现同样的目的），然后在内部最终会导致 `terminate/1` 回调函数被调用，该函数接收进程状态作为参数，并执行一切关闭 `server` 所必需的特定的代码。在我们的例子里，我们把例子尽可能地进行了简化。然而，我们也可以设计成在 `terminate/1` 中自动终止所有分配了频率的 `client` 进程：

```
stop(Name) ->                                     % server.erl
    Name ! {stop, self()},
```



```

receive {reply, Reply} -> Reply end.

call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.

reply(To, Reply) ->
    To ! {reply, Reply}.

```

为了确保接口不变，新实现的 frequency 模块中导出的函数和之前完全一样：

```

stop()          -> server:stop(frequency).           % frequency.erl
allocate()      -> server:call(frequency, {allocate, self()}).
deallocate(Freq) -> server:call(frequency, {deallocate, Freq}).

```

这些函数会发送普通请求、stop 消息给 server。当进程接收到消息后，frequency 模块中相关的回调函数将会被调用。例如收到 stop 消息后，terminate/1 回调函数会被调用。该函数接受进程状态作为参数，而它的返回值将会被发送给 client，最后成为 stop/1 的返回值：

```

loop(Mod, State) ->                                     % server.erl
    receive
        {request, From, Msg} ->
            {NewState, Reply} = Mod:handle(Msg, State),
            reply(From, Reply),
            loop(Mod, NewState);
        {stop, From} ->
            Reply = Mod:terminate(State),
            reply(From, Reply)
    end.

```

而调用请求时，则是 handle/2 回调函数被调用。调用时涉及两个参数，第一个是消息本身，它绑定到了变量 Msg 上；第二个是进程状态，它绑定到了变量 State 上。通过对 Msg 进行模式匹配，将会挑选出 handle 函数的合适的分句来处理该消息。handle 回调函数必须返回一个格式为 {NewState, Reply} 的元组，其中 NewState 包含了更新后的频率列表，而 Reply 则是发回给 client 的回应。看一看 allocate/2 的实现。它返回的正好是一个元组，其中第一个元素是更新后的进程状态，而第二个元素则是 {ok, Frequency} 或是 {error, no_frequency}。

◀ 72

loop/2 的 receive 的第一个分句把 handle/2 的返回值作为回应使用 reply/2 发回给 client，并使用新的状态继续循环，等待下一个请求：

```

terminate(_Frequencies) ->                               % frequency.erl

```

ok.

```
handle({allocate, Pid}, Frequencies) ->
    allocate(Frequencies, Pid);
handle({deallocate, Freq}, Frequencies) ->
    {deallocate(Frequencies, Freq), ok}.

allocate({[], Allocated}, _Pid) ->
    {[[], Allocated], {error, no_frequency}};
allocate({[Freq|Free], Allocated}, Pid) ->
    {[Free, [{Freq, Pid}|Allocated]], {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated = lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free], NewAllocated}.
```

同样的过程也适用于 `deallocate` 请求。频率被释放，`handle/2` 调用返回一个元组，其中包含新的状态（来自调用 `deallocate/2` 后的返回值），以及原子 `ok`，然后这个元组作为回应发回给 `client`。

至此，我们已经把频率服务器例子分割成了两部分，一部分是通用的库模块——我们称其为 `server`，另一部分是专用的模块——我们称其为 `frequency` 模块。这些内容，如果你理解了，那么也就理解 Erlang 的行为模式了。本质上来说，就是把代码分为通用的和专用的两部分，然后把通用的部分打包为可重用的库，这样就可以尽可能地把复杂的部分隐藏好，无须开发人员操心。我们故意把例子设计得如此简单，就是为了突出重点，因此忽略了各种特殊的边界情况——这些情况在真实的行为模式库中会做恰当处理。下一节我们会介绍这方面的细节，而且之后介绍各个行为模式库的时候也会介绍到。

消息传递：冰山之下

并发编程并不容易。你需要处理竞态条件、死锁、临界区以及许多边界情况。话虽如此，但实际上却鲜有 Erlang 开发人员抱怨并发编程，更不用说抱怨这些问题了。原因很简单：OTP 框架的设计使得这些问题中的大多数不再成为一个问题。在这一章，我们从一个特定的 `client-server` 系统中抽取出了通用的代码，并且完成这一任务的同时我们还让例子保持尽可能简单。在真实场景中有许多错误情形需要处理，这些错误情形通常会被行为模式库模块为我们在幕后处理掉，下一章里我们将会介绍这方面的内容。值得强调的是，这些处理工作并不需要程序员介入。竞态条件，特别是在多核体系结构之上的时候，已经变得越发常见，找出此类问题的较好方法是使用合适的建模与测试工具，例如 `Concuerror`、`McErlang`、`PULSE` 和 `QuickCheck` 等。

说了这么多，接下来我们还是看一个例子，了解一下行为模式库在帮助我们处理各种棘手问题方面所做的那些事。对于一个经验不够丰富的开发人员来说，很多问题他们一开始并不能意识到。我们直接使用前面例子里的 `call/2` 函数，然后随着讲解我们再扩充它：

```
call(Name, Message) ->
  Name ! {request, self(), Message},
  receive
    {reply, Reply} -> Reply
  end.
```

```
reply(Pid, Reply) ->
  Pid ! {reply, Reply}.
```

我们发送格式为 `{request, Pid, Message}` 的消息到 `server`，然后等待格式为 `{reply, Reply}` 的回应。如图 3-8 所示，当接收到回应后，如何确保回应就是真的来自 `server`，而不是来自另外某个也采用了相同协议的进程？

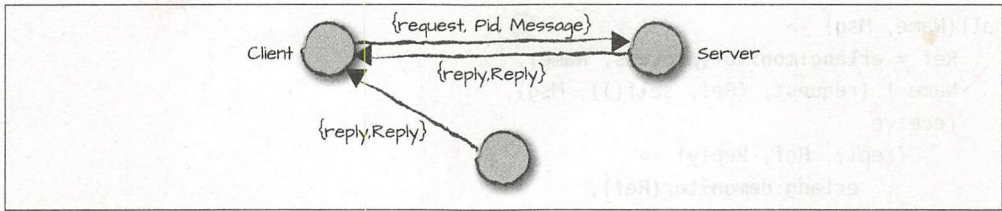


图 3-8: 消息竞态条件

就当前实现来说，我们无法区分。解决这一问题的办法就是使用“引用”（reference）。通过调用 `make_ref()` BIF 可以创建一个具有唯一性的引用，然后将其添加到消息里，并在回应时也带上，这样就可以认定该消息确实是对我们发出的那个请求的回应，而不会是某条碰巧也遵循相同协议的无关消息。加上引用后，我们的代码看起来就像这样^{注1}：

```
call(Name, Msg) ->
  Ref = make_ref(),
  Name ! {request, {Ref, self()}, Msg},
  receive {reply, Ref, Reply} -> Reply end.
```

```
reply({Ref, To}, Reply) ->
  To ! {reply, Ref, Reply}.
```

注意，`receive` 分句里的 `Ref` 是已经绑定过的变量，这样就可以保证我们匹配到的回应

注1 为使 `stop` 调用能够工作，其实代码还要做一些小改动。但在这个例子里我们省略了。

必然对应的是我们先前发出的消息。好的，这个方案解决了一个问题，但这就足够了？如果 server 在我们发送请求前恰巧就崩溃了呢？如果 Name 此时对应的是一个注册进程的别名，那么不会有问题，因为向一个不存在的注册进程发送消息将会导致 client 进程终止。但如果 Name 对应的是 pid，那么消息将会丢失，而 client 则会永远停顿在调用函数的 receive 分句处。或者换一种类似的情况，如果恰巧 server 在收到消息和发送回应之间崩溃了又会发生什么呢？触发崩溃的既可能是我们发送的请求，也可能是其他 client 发送的请求。在这种情况下，即使 Name 对应的是注册进程的别名也没用，因为当发送消息时 server 进程是存活的。

解决方案就是“监视”server。要做到这一点，需使用 monitor BIF 而不是 link，因为 link 是双向的，可能会导致一些副作用（比如 server 在收到该请求时恰好又准备杀掉 client 进程）。虽然 client 监视着 server 的终止，但是 client 自身的终止却不会反过来影响到 server。使用 monitor BIF 会返回一个唯一的引用，所以我们没必要再用 make_ref() BIF，直接使用 monitor 的引用来标记我们的消息就够了：

```
call(Name, Msg) ->
    Ref = erlang:monitor(process, Name),
    Name ! {request, {Ref, self()}, Msg},
    receive
        {reply, Ref, Reply} ->
            erlang:demonitor(Ref),
            Reply;
        {'DOWN', Ref, process, _Name, _Reason} ->
            {error, no_proc}
    end.
```

我们做的这些已经涵盖全部出错的可能了吗？没有。当我们去监视其他进程的时候，我们又把自己暴露在了另一种竞态条件下。考虑下面的事件序列：

1. client 监视 server。
2. client 向 server 发送了一个请求。
3. server 接收到请求并进行了处理。
4. server 向 client 发送了回应。
5. server 开始处理另一个请求，而不幸的是，它因此而崩溃了。
6. client 因而收到了一条来自 monitor 的 DOWN 消息。
7. client 从邮箱中抽取出了来自 server 的回应。
8. client 解除对 server 的监视（注意此刻 server 早已不存在）。

client 邮箱中会存在一条 DOWN 消息，由于其引用无法匹配，我们卡住了。那么，发生这种状况的几率有多大呢？你真的认为有人会想为这样的场景设计对应的测试用

例 (test case) —— 让 server 在发送完给 client 的回应后, 但 client 还没能执行到 `erlang:demonitor/2` 调用的这一时刻精准地崩溃掉——吗? 尽管这是一种极端的边界状况, 我们还是需要处理掉残留的 DOWN 消息, 否则会产生内存泄漏。所以我们通过在调用 `demonitor/2` 时传入 `[flush]` 选项来确保凡是属于该 monitor 的 DOWN 消息都不会残留在邮箱中。

好了, 这一次总该完美了吧? 没有——如果 Name 对应的不是一个已经注册的进程别名呢? client 向不存在的注册进程发送消息而导致的异常我们都得捕获。不过我们并不关心具体捕获到了什么样的异常——如果我们关心的话, 就会使用 `try-catch` 语句而不是 `catch` 语句, 因为如果 server 不存在的话, `monitor/1` 会发送一条 DOWN 消息。现在我们新的代码看起来会是这样:

```
call(Name, Msg) ->
  Ref = erlang:monitor(process, Name),
  catch Name ! {request, {Ref, self()}, Msg},
  receive
    {reply, Ref, Reply} ->
      erlang:demonitor(Ref, [flush]),
      Reply;
    {'DOWN', Ref, process, _Name, _Reason} ->
      {error, no_proc}
  end.
```

不幸的是, 哪怕改了这么多次, 还是不够。如果进程 A 向 B 发起同步调用的同时, B 也发起了对 A 的调用, 那会怎么样? 此处的“同步调用”指的是一种 Erlang 消息交互过程——发送消息的进程会等待回应, 并且消息和回应都各自以同步消息的方式发送。进程 A 一发出请求后就立刻进入 `receive` 分句等待着一一条能匹配引用的回应出现, 而 B 的做法也一模一样。回答我们最初的问题, 就我们编写的这份代码而言, 如果两个进程相互同步调用对方, 那么将导致死锁 (deadlock)。虽说死锁是由于设计阶段的缺陷导致的, 但是运行中的系统也会出现这样的状况, 这样一来我们就需要准备好一套 (最好是通用的) 恢复机制以便应对。解决死锁最简单的办法就是在 `receive` 语句中使用超时机制来终止进程。在下一章里, 我们会介绍死锁和超时的更多细节, 并展示 OTP 是如何解决这一问题的。

总结

76

在本章, 我们涵盖了并发设计模式背后的一些原则, 介绍了行为模式库的概念。希望我们已经让大家认识到行为模式库的重要性与强大性, 因为理解这些知识是进一步认识 OTP 背后原理的基础, 可以说它们凝聚了工程师们在面向进程编程 (process-oriented

programming) 领域数十年的心血,减轻了如今的开发人员肩上的负担,减少了需要编写的代码量,并保证各种极端特殊情形都能够被一致、高效、正确地处理。请扪心自问:我们在本例中讨论了不少特殊情形的处理,其中有多少是你在第一轮编写中就能考虑周全的呢?你的同事们又如何呢?想象一下,如果你要测试和维护的系统中,每个人都在重新发明轮子,把各种并发情况和边界条件的处理都按照个人的有限理解去处理,那会有多可怕!而标准 OTP 行为模式的价值恰恰在于它为你兜底处理了这许许多多的问题,这也正是为什么你应当使用它的原因。

如果你有时间的话,学 Erlang 时不妨选一个客户端 - 服务器小程序练练。可以是键值存储方面的程序,也可以是聊天服务程序,或者其他某种接收并处理请求的程序。如果你手边没有任何例子可用,那就使用 *Erlang Programming* 一书中 ETS 和 DETS 一章里 mobile subscriber database 的例子。你可以从书籍作者的 GitHub 仓库中下载相关的代码。

另一个有用的练习方式是扩展调用函数,增加 `after` 分句,在其中使得进程退出,原因设置为 `timeout`。创建一个新函数:

```
call(Name, Message, Timeout)
```

在这个函数里, `Timeout` 可以是一个单位为毫秒的整数,或者是原子 `infinity`,用户 can 以此设置超时时间。保留 `call/2` 调用,设置默认值为 5 秒。如果 `server` 没有在指定的超时时间内响应,就让 `client` 进程非正常终止,将原因设置为 `timeout`。进程结束前不要忘了清理,因为 `exit` 信号可能会被 `server` 库代码中的某个 `try-catch` 表达式捕获。

接下来是什么

在后面的章节中,我们将介绍一系列库模块,它们为我们提供了各种 OTP 行为模式。我们先介绍 `gen_server` 库,然后以相同的风格依次介绍 `gen_fsm` (有限状态机)、`gen_event` (事件处理器)、`supervisor` (监督者)以及 `application` (应用)。我们还会涉及死锁 (deadlock)、超时 (timeout) 以及各种错误情况的处理——包括分布式环境下的,以及由于消息不匹配而带来的等。这些主题在我们介绍各个库时都会讨论到。

通用型服务器

当我们把无线频率分配器（radio frequency allocator）划分为由通用的和专用的两类模块构成，并且探讨了一些并发环境下可能出现的边界情况的处理方法之后，你应该已经认识到，以后当你再次需要实现 client-server 行为模式的时候，已经没必要再把这个流程走一遍了。在本章，我们介绍 `gen_server` OTP 行为模式，它是一个库模块，提供了所有通用型 client-server 所需功能的同时，还处理了大量的边界情况。`gen_server` 是最常用的行为模式，它是基础中的基础，其余所有行为模式都可以基于它来实现——事实上早期的 OTP 框架就是这么做的。

`gen_server`

`gen_server` 模块实现了我们在第 3 章里分析出的 client-server 行为模式。该模块是标准库应用的一部分，与 Erlang/OTP 一同分发。它包含了一些通用的代码，这些代码能够通过一组回调函数作为接口，与相应的回调模块组合。所谓的回调模块，在我们的例子里，是指与频率服务器紧密相关，由相应程序员来实现的那些部分。这样的回调模块必须导出一系列名称和类型都符合约定的函数，这样一来它们的输入以及返回值才满足行为模式的需要。

如图 4-1 所示，行为模式和回调模块二者的函数都是在同一个 server 进程内执行的。换言之，进程是在 `gen_server` 模块内循环执行的，根据需要再调用回调模块中的函数。

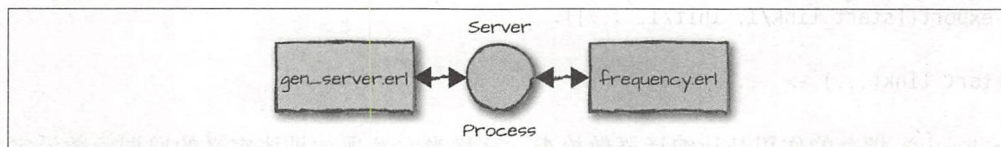


图 4-1：回调模块与行为模式模块

78 **gen_server** 库模块提供了一些函数，可以用来启动和停止 **server**。你可以提供回调代码来实现系统的初始化，而且，当进程正常或者非正常终止时，都可以先调用你的回调模块中的函数来完成状态清理操作再终止。尤其值得注意的是，你不再需要向你自己的进程发送消息了。**gen_server** 把消息传递功能包装成两个函数——一个用来发送同步式消息，另一个用来发送异步式消息。这二者将负责处理第 3 章里我们曾讨论过的所有特殊情况，以及许多我们都没有意识到的可能导致故障或者引发竞态条件的情况。除此之外，**gen_server** 模块还内置了与软件更新有关的功能，借助这些功能你可以暂停你的进程，并将当前版本系统中的数据迁移到下一个版本的系统中。**gen_server** 还提供超时机制，在 **client** 侧发送请求，在 **server** 侧当超过预定时间没有接收到消息时也提供相应的功能。

我们现在介绍完了使用 **gen_server** 时会用到的所有回调函数。它们包括：

- 当调用 **gen_server:start_link/4** 函数创建 **server** 进程时，用于初始化的 **init/1** 回调函数。
- **handle_call/3** 回调函数用于处理调用 **gen_server:call/2** 后发往 **server** 的同步式请求。当请求处理完毕后，**call/2** 返回的值就是 **handle_call/3** 计算后返回的值。
- 异步式请求则是由 **handle_cast/2** 回调函数处理的。这种请求来源于 **gen_server:cast/2** 调用，它会发送一条消息给 **server** 进程然后立刻返回。
- 当 **server** 回调函数中任何一个返回 **stop** 消息时，将导致 **server** 进程终止，终止前会调用 **terminate/2** 回调函数。

我们很快会进一步学习这些函数的细节，包括它们的参数、返回值以及与其有关的回调函数等。在此之前，我们得先介绍一下模块指令（**module directive**）。

behavior 指令

当实现 OTP 行为模式时，我们需要在模块声明中添加 **behavior** 指令。

79

```
-module(frequency).  
-behavior(gen_server).  
  
-export([start_link/1, init/1, ...]).  
  
start_link(...) -> ...
```

behavior 指令的作用是让编译器做检查，这样当它发现如果该定义的回调函数没定义，或者虽然定义了却没导出，或者元数（**arity**）不正确时产生一些警告信息。另外，

`dialyzer` 工具也会使用这些声明来检查类型不一致方面的问题。更重要的是, `behavior` 指令对于那些不得不支持、维护、调试你的代码的可怜人^{注1} 尤其重要(那时的你早就转往其他更加刺激有趣的项目去了,而他们还接手这一切)。他们一看到这些指令,马上就能明白你用了 `gen_server` 模式。如果他们想看看 `server` 初始化过程,就会跳到 `init/1` 函数处。如果他们想看看 `server` 终止时的清理过程,就会跳到 `terminate/3` 函数处。相较于那种每个公司、每个项目、每个开发人员反反复复重新实现 `client-server` 而且还常常可能带有各种错误的过去来说,这真的是一个巨大的进步。不需要再把时间浪费在去理解各种各样的框架上,这使得那些阅读代码的人可以将注意力聚焦到重要的部分。

应该拼作 `behavior` 还是 `behaviour`

你可能已经注意到,我们在回调模块里添加的 `behavior` 指令是美式拼法。英国朋友们别难过。在定义 `behavior` 指令时,美式的“`behavior`”和英式的“`behaviour`”拼写都是支持的:

```
-behavior(tcp_wrapper).  
-behaviour(tcp_wrapper).
```

当使用 `behavior_info/1` 定义回调函数时也一样支持两种拼写。然而很久以前,如果你没有遵循英式拼写,骄傲地坚持不打那个额外的字母,将会在编译回调模块时得到一条警告信息“`unknown behaviour`”。许多人困惑地花费了大把时间才找到问题所在。

在我们的示例代码中,编译器针对 `-behavior(gen_server)` 指令产生了一条警告信息,因为我们漏了 `code_change/3` 函数,关于这个回调函数我们将在第 12 章讨论版本更新的时候再介绍。除了这条指令,我们有时候还会使用另一条相对次要的、可选的指令 `-vsn(Version)` 来实现代码升级(以及降级)过程中对模块版本号的追踪。详细的情况在第 12 章会介绍。

启动一个 `server`

80

了解了模块指令的知识后,我们来启动 `server` 吧。启动 `gen_server` 以及其他 OTP 行为模式不能直接通过调用 `spawn BIF` 完成,应该使用这个更加专门的函数,它内部所做的远不仅仅是分裂新进程那么简单:

```
gen_server:start_link({local,Name},Mod,Args,Opts) ->  
    {ok, Pid} | ignore | {error, Reason}
```

注1 善待这些人,毕竟风水轮流转,没准哪天你也和他们一样呢。

`start_link/4` 函数接受 4 个参数。第一个参数告诉 `gen_server` 模块按 `Name` 指定的别名把进程注册到本地。`Mod` 是回调模块的名字，其中应当包含各种回调函数以及与 `server` 相关的代码。`Args` 是一个 Erlang 数据项 (term)，会被传递给相应的 `callback` 函数用于初始化 `server` 状态。`Opts` 是一个列表，其中包括与进程和调试相关的各种选项，我们会在第 5 章介绍。作为入门，我们将它简化一下，`Opts` 处传入一个空列表就行。如果进程已经以相同的 `Name` 别名注册过了，那么将返回 `{error, {already_started, Pid}}`。请瞪大眼睛看清到底是哪个进程执行了哪个函数。观察图 4-2 你会注意到，`server` 绑定到的 `Pid` 进程实际是由 `supervisor` 启动的。`supervisor` 由双圆圈表示，因为它会捕获 `exit` 信号。

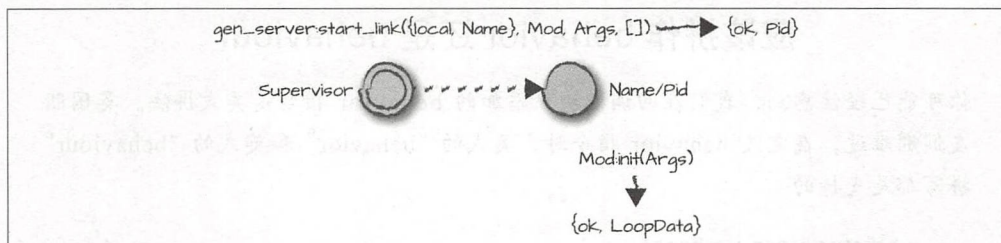


图 4-2: 启动一个 `gen_server`

当 `gen_server` 进程分裂好后，会为其注册好 `Name` 指定的别名，然后紧接着调用回调模块 `Mod` 中的 `init/1` 函数。不管 `init/1` 函数自身需要与否，都会接收到一个 `Args` 作为参数——正是当初调用 `start_link` 函数时传的第三个参数。如果 `init/1` 并不需要参数，那么可以通过使用“无所谓 (don't care)”变量来忽略。记住，`Args` 可以是任意的 Erlang 数据项，并非必须是列表。



如果 `Args` 是一个列表 (可能为空)，那么这个列表传给 `init/1` 时也是保持原样作为列表传递，不会转而去调用不同元数的 `init` 函数。例如，如果你传的是 `[foo, bar]`，则 `init([foo, bar])` 将被调用，而不是 `init(foo, bar)`。这是一个常见的误解，特别是开发人员从 Erlang 转向 OTP 的过程中，会因为把 `spawn`、`spawn_link` 函数与用于启动行为模式的 `start`、`start_link` 函数搞混而导致错误。

81 `init/1` 回调函数负责初始化 `server` 状态。在我们的例子里，其实就是创建一个变量，其中包含两个列表，一个代表可用频段，另一个代表已分配频段。

```

start() ->                                     % frequency.erl
    gen_server:start_link({local, frequency}, frequency, [], []).

init(_Args) ->
    Frequencies = {get_frequencies(), []},
  
```



```
{ok, Frequencies}.
```

```
get_frequencies() -> [10,11,12,13,14,15].
```

如果成功，`init/1` 回调函数要返回 `{ok, LoopData}`。如果启动失败，但你不想影响到由当前 supervisor 启动的其他进程，就返回 `ignore`。如果想影响到，则返回 `{stop, Reason}`。我们会在第 8 章介绍 `ignore`，在本章后面介绍 `stop`。

在我们的例子里，`start_link/4` 传递了空的列表 `[]` 给 `init/1`，而 `init/1` 最终用 `_Args don't care` 变量忽略了这一参数。我们本来可以传递其他任意 Erlang 数据项的，但考虑到阅读代码的人能清晰理解此处实际上并不需要任何参数，所以我们没有那么做。除了空列表，也可以用原子 `undefined` 或者空元组 `{}`，都是受欢迎的选择。

通过在 `Opts` 列表传入 `{timeout, Ms}` 作为选项，我们可以指定允许花费多少时间（毫秒）来启动我们的 `gen_server`。如果花费的时间超过此设定，`start_link/4` 将返回元组 `{error, timeout}` 并且 `behavior` 进程不会启动。没有异常会被抛出。我们会在第 5 章介绍更多的选项细节。

启动 `gen_server behavior` 进程的操作是同步式的。只有当 `init/1` 回调函数返回 `{ok, LoopData}` 给 `server` 循环后，`gen_server:start_link/4` 函数才会返回 `{ok, Pid}`。清楚地认识到 `start_link` 的这种同步式特点是至关重要的，其对于重现启动顺序非常关键。当我们针对某些启动阶段发生的故障进行排查时，这种能够确定性地重现错误的能力可以说非常重要。是的，你可以异步式地启动所有的进程，然后再逐个检查是否都已正确启动。但如果改变了多核体系上的调度器（scheduler）实现，或者修改了其配置参数，又或者部署到了不同的硬件或操作系统上，甚至仅仅是网络连接性状态的改变，都可能导致进程间初始化、启动顺序的改变。如果一切顺利，倒也不会在这种充满不确定性的异步启动过程中遇到什么问题，可一旦竞态条件显现，再想试图搞清到底是哪个部分在哪个阶段出了问题——特别是在生产环境下——心脏承受能力不强的人我劝你别试。`start_link` 中采取的同步式启动方案通过其简单性——一个进程正确启动了之后才会启动下一个——清晰地为我们提供了保障，使得我们对于单个节点上发生的启动错误能够确定性地重现。如果启动错误是受外部因素的影响，例如网络、外部数据库或者底层硬件、操作系统，试着把它们也包含进来。如果确定性还不够帮助你解决问题，试着进一步控制启动过程，把其中你怀疑可能产生问题的元素都移除试试。

82

消息传递

启动了 `gen_server` 并初始化好其循环所用的数据后，现在该看看通信了。正如你从前一章学到的那样，其实并不需要使用 `!` 操作符来发送消息。OTP 通过函数式的接口提供了

更高级别的抽象。`gen_server` 模块导出了一些函数，允许我们既可以同步式发送消息，也可以异步式发送消息，为程序员隐藏了并发编程和错误处理背后的复杂性。

同步式消息传递

尽管 Erlang 将异步式消息传递作为语言的一部分内置好了，但我们并不会因此受限而无法基于此原语实现同步式调用。`gen_server:call/2` 函数正是这一功能的实现者。它发送一个同步式消息 `Message` 给 `server` 并在 `server` 忙于在回调函数中处理请求时等待其回应 `Reply`。然后 `Reply` 作为返回值返回。请求消息与回应遵循一定的协议，并包含了一个唯一的标记（用于引用），使得请求消息可以与回应对应。我们不妨更进一步来看看 `gen_server:call/2` 函数：

`gen_server:call(Name, Message) -> Reply`

`Name` 可以是 `server` 的 `pid` 或者 `server` 进程注册过的名称。`Message` 则是一个 Erlang 数据项，将被作为请求的一部分发给 `server`。请求作为一个普通的 Erlang 消息，接受后会存储在邮箱里，并按顺序处理。一旦接收到一条同步请求，回调模块中的 `handle_call(Message, _From, LoopData)` 回调函数将会被调用。其第一个参数正是调用 `gen_server:call/2` 时传入的那个 `Message`。第二个参数 `_From` 包含了一个唯一的请求标识符以及与 `client` 有关的信息。我们暂时忽略它，把它绑定到一个 `don't care` 变量上。第三个参数是 `init/1` 回调函数当初返回的 `LoopData`。你应该可以看懂图 4-3 里面的流程了。

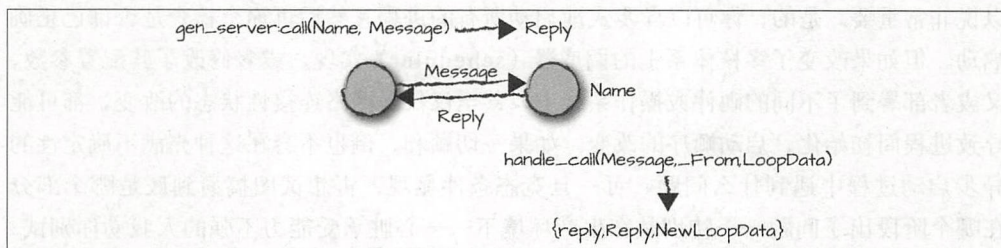


图 4-3：同步式消息传递

83 `handle_call/3` 回调函数包含了与处理请求相关的所有代码。实践中比较好的做法是，针对每种请求使用模式匹配对应到单独的 `handle_call/3` 分句处理，而不是使用一个 `case` 语句去分拆。在函数分句中，我们会执行一切与该请求有关的代码，完毕后，返回一个形式如 `{reply, Reply, NewLoopData}` 的元组。回调模块使用原子 `reply` 告诉 `gen_server`，第二个元素 `Reply` 需要被发回给 `client` 进程，成为 `gen_server:call/2` 请求的返回值。第三个元素 `NewLoopData`，是回调模块的新状态，将被 `gen_server` 在下一轮“接收 - 求值”尾递归的迭代中传入。如果 `LoopData` 在函数体内执行过程中没有变化，我们

直接在回应的元组中返回原始值即可。gen_server 不会查看或者操作此数据的内容，仅仅是存储起来而已。一旦回应元组发回 client，server 于是又准备好处理下一个请求了。如果进程邮箱中没有消息可处理，server 将被挂起等待新请求进入。

在我们的频率服务器例子里，为了分配频率需要做同步式调用，在调用返回的结果里需要包含分配的频率。为了处理这样的请求，我们调用内部函数 allocate/2，这个函数你可能还记得返回 {NewFrequencies, Reply}。NewFrequencies 是一个元组，其中包含了已分配的频率列表和可用的频率列表，而 Reply 则是元组 {ok, Frequency} 或者 {error, no_frequency}：

```
allocate() ->                                     % frequency.erl
    gen_server:call(frequency, {allocate, self()}).

handle_call({allocate, Pid}, _From, Frequencies) ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    {reply, Reply, NewFrequencies}.
```

完成后，被 client 进程调用的 allocate/0 函数返回 {ok, Frequency} 或者 {error, no_frequency}。而更新后的包含了可用和已分配频率的循环数据将被存储在 gen_server 的“接收 - 求值”循环中等待下一个请求。

异步式消息传递

84

如果 client 需要发送消息给 server 但不期待回应，则可以使用异步式请求。这要靠 gen_server:cast/2 库函数来完成：

```
gen_server:cast(Name, Message) -> ok
```

Name 可以是 pid 或者本地注册的 server 进程别名。Message 是 client 想要发给 server 的数据项。一旦 cast/2 调用发送了请求便会立刻返回原子 ok。在 server 端，请求被存储到进程邮箱里并被依次处理。当其被收取时，Message 被传递给开发者在回调模块里实现的 handle_cast/2 回调函数。

handle_cast/2 回调函数接受两个参数。第一个是发自 client 的 Message，第二个是之前由 init/1、handle_call/3 或者 handle_cast/2 回调返回的 LoopData。你可以在图 4-4 里看到这一切。

handle_cast/2 回调函数必须返回一个格式为 {noreply, NewLoopData} 的元组。NewLoopData 将会作为参数在下一次处理同步式或异步式请求时被传入。

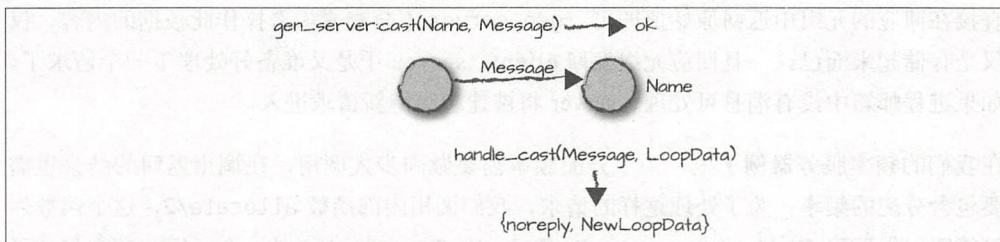


图 4-4: 异步式消息传递

在一些应用程序里，client 函数返回一个硬编码的值，一般随回调模块里执行时的副作用不同而不同，通常是原子 ok。这类函数可以被实现为异步式调用。在我们的频率服务器例子里，你有没有注意到 `frequency:deallocate(Freq)` 总是返回原子 ok？在此处，我们其实并不关心请求是否因为 server 目前忙于处理其他请求而延迟了处理当前请求，这一特点恰好使得此处非常适合作为使用 `gen_server` 的 `cast` 的一个例子：

```

deallocate(Frequency) ->                                % frequency.erl
    gen_server:cast(frequency, {deallocate, Frequency}).

handle_cast({deallocate, Freq}, Frequencies) ->
    NewFrequencies = deallocate(Frequencies, Freq),
    {noreply, NewFrequencies};
  
```

85 client 函数 `deallocate/1` 发送一个异步式请求给 `gen_server`，然后立即返回原子 ok。请求会被 `handle_cast/2` 函数拾取，其第一个参数对消息进行模式匹配 `{deallocate, Frequency}`，而第二个参数则把循环数据绑定到 `Frequencies` 变量。在函数体内，调用了辅助函数 `deallocate/2`，将 `Frequency` 从已分配频率列表移到可用频率列表。`deallocate/2` 的返回值被绑定到了变量 `NewFrequencies`，作为新的循环数据在 `noreply` 控制元组中一同返回。

注意，我们说过，只有某些应用程序中的 client 函数会在调用带有副作用的函数时忽略返回值。而像 ping 某个 server 确定其是否存活之类，则应该用 `gen_server:call/2`，这样当 server 已经终止时，将触发异常。另外还可以根据调用返回的延迟判断 server 处理本次请求和发送回应期间是否处于重负载状态。再举一个使用同步式调用的例子，例如当需要节流（throttle）请求并控制消息发往 server 的频率的情况。我们在第 15 章将讨论与节流相关的内容。

和纯 Erlang 一样，如果要从模块外使用，`call` 和 `cast` 应当被抽象为函数式 API。这样你将拥有极大的灵活性，可以改变协议、对函数调用者隐藏私有的与实现相关的信息等。把 client 函数放在与进程相同的模块内，这样有助于理解消息流，无须在模块间跳来跳去。

其他消息

OTP behavior 是通过 Erlang 进程实现的。因此，虽然理想情况下通信应该是基于 `gen_server:call/2` 和 `gen_server:cast/2` 中定义的协议进行的，但现实中也有例外。只要进程 `pid` 或者注册的别名是已知的，你就无法阻止用户使用 `Name!Message` 方式发送消息。在某些情况下，原始的 Erlang 消息甚至是 `gen_server` 获知相关信息的唯一途径。举一个例子，如果 `server` 与另一些进程或端口相链接，而且又调用过 `process_flag(trap_exit, true)` BIF 设定为捕捉这些进程、端口的退出信号，则该 `server` 可能会收到 `EXIT` 信号消息。另外，进程、端口、套接字之间的通信同样是基于消息传递的。除此之外，当我们正使用进程监视器（process monitor）监视分布式节点，或者正与某些遗留的、不兼容的 OTP 代码相互通信时，同样会涉及这些问题。

这些例子都表明，我们的 `server` 会收到某些消息，它们不遵循 OTP 内部所定义的消息协议。如果你正在使用某些特性，其会向你的 `server` 产生消息，那么你的 `server` 代码就必须能够处理这些消息。`gen_server` 提供了一个回调函数，它负责处理所有此类消息。它就是 `handle_info(_Msg, LoopData)` 回调函数。当此函数被调用的时候，它必须返回元组 `{noreply, NewLoopData}` 或者 `{stop, Reason, NewLoopData}`，后者代表停止。

86

```
handle_info(_Msg, LoopData) ->                                % frequency.erl
    {noreply, LoopData}.
```

实践中常见的做法是，即使你没有打算处理任何消息，也把这个回调函数带上。否则，如果向 `server` 发送了一条不兼容 OTP 的消息将导致出现运行时错误，导致 `server` 终止，原因是本应调用回调模块中的 `handle_info/2` 函数来处理该消息，却由于函数未定义而导致了出错。

我们已经让我们的频率服务器例子足够简单了。对于任何收到的消息，我们一概忽略，返回 `noreply` 元组，且其中的 `LoopData` 没有修改过。如果你确定不应接收到任何非 OTP 的消息，你可以记录下该消息，将其视为错误。如果想每次与 `server` 相链接的其他进程非正常终止时就输出一条错误信息，代码大概是这样的（假设本问题中的 `server` 已经被设定为捕捉退出信号）：

```
handle_info({'EXIT', _Pid, normal}, LoopData) ->
    {noreply, LoopData};
handle_info({'EXIT', Pid, Reason}, LoopData) ->
    io:format("Process: ~p exited with reason: ~p~n", [Pid, Reason]),
    {noreply, LoopData};
handle_info(_Msg, LoopData) ->
    {noreply, LoopData}.
```



OTP 的缺点包括各种行为模式之间的过度分层, 以及由通信协议带来的额外数据开销。这些都会影响性能。为了能节省这些调用占据的几微秒, 开发人员知道可以绕过 `gen_server:cast` 函数而直接使用 `Pid!Msg` 方式, 甚至更糟的, 直接把 `receive` 语句放到回调函数里用来接收消息。不要这么做! 这会让你的代码难以调试、支持和维护, 失去 OTP 为你带来的各种优势, 还会使得本书的作者们不再喜欢你。如果你需要在微秒级别上节约时间, 先搞清楚程序性能在实际环境下度量后是否真的不够快, 然后再考虑优化。

未处理的消息

Erlang 从进程邮箱接收消息时使用的是选择性接收。这允许我们抽取特定消息的同时把其他消息留下不处理, 但带来了内存泄漏的风险。如果某类消息永远不会被取出将会发生什么? 在只使用 Erlang 但不使用 OTP 的情况下, 消息队列将会变得越来越长, 这导致要想使用模式匹配成功地拾取一条消息时需要遍历的消息数量越来越多。这样不断增长的消息队列将会暴露出来, 因为遍历邮箱会导致 Erlang VM 的高 CPU 占用, 然后最终耗尽内存, 并且可能会被 *heart* 部件重启, 关于 *heart* 部件我们会在第 11 章介绍。

如果我们使用的是纯 Erlang, 则这一切都是合规的, 但 OTP behavior 采取了一种不同的方式处理。消息的处理顺序与其接收顺序相同。启动你的频率服务器, 并且试着发过去一条不会被处理的消息:

```
1> frequency:start().
```

```
{ok,<0.33.0>}
```

```
2> gen_server:call(frequency, foobar).
```

```
=ERROR REPORT==== 29-Nov-2015::18:27:45 ===
```

```
** Generic server frequency terminating
```

```
** Last message in was foobar
```

```
** When Server state == {data,[{"State",
```

```
    {{available,[10,11,12,13,14,15]},  
     {allocated,[]}}}]}
```

```
** Reason for termination ==
```

```
** {function_clause,{{frequency,handle_call,
```

```
    [foobar,
```

```
    {<0.44.0>,#Ref<0.0.4.112>},
```

```
    [[10,11,12,13,14,15],[]]},
```

```
    [{file,"frequency.erl",{line,63}}],
```

```
    {gen_server,try_handle_call,4,
```

```
    [{file,"gen_server.erl",{line,629}}],
```

```
    {gen_server,handle_msg,5,
```

```
    [{file,"gen_server.erl",{line,661}}],
```

```
    {proc_lib,init_p_do_apply,3,
```

```
    [{file,"proc_lib.erl",{line,240}}]}}
```


这可能不是你想要的效果。频率服务器由于 `function_clause` 运行时错误而终止了，输出了一份错误报告。^{注1} 因为调用某个函数的时候，该函数的若干分句中必须有一个是匹配的，如果全都不匹配就会引发运行时错误。当执行 `gen_server` 上的 `call` 或者 `cast` 时，消息一定会在 `gen_server` 循环过程里被从邮箱中取出，然后相应地调用 `handle_call/3` 或者 `handle_cast/2` 回调函数。在我们的例子里，`handle_call(foo, _From, LoopData)` 不匹配任何一个分句，导致了我們刚才看到的函数分句错误。同样的规律也适用于 `cast`。

如何才能避免这样的错误呢？一个办法是使用 `catch-all`，对于未知消息一概匹配到一个 `don't care` 变量上然后忽略掉。这一做法是否适用，取决于具体的应用程序，可能是答案，也可能不是答案。在处理端口、套接字、链接、监视器以及监视分布式节点等可能出现忘记处理某些应用所需的消息时，`catch-all` 的做法可能是符合规范的。但处理 `call` 和 `cast` 的时候则不同，所有的请求都应该来自 `behavior` 回调模块并且任何未知消息都应该在测试早期被找出来。

如果有疑虑，收到未知消息时，不要采取防御式手段，而是直接让 `server` 终止。把这样的终止当作 `bug` 对待，要么考虑如何正确处理这样的消息，要么从消息的源头进行修正。当你决定忽略未知消息时，记得输出到日志记录。

同步客户端

考虑这样的场景，两个 `client` 同时各自发了一个同步式请求给某个 `server`，该 `server` 无法单独地依次回应这两个请求，而是为了回应第一个请求却必须等待收到第二个请求才行，那么在这种情况下会发生什么？我们通过图 4-5 展示了这一过程。出现这种状况可能是因为需要进行某方面的同步，或者是由于 `server` 要做出回应必须集齐两端请求的数据等。

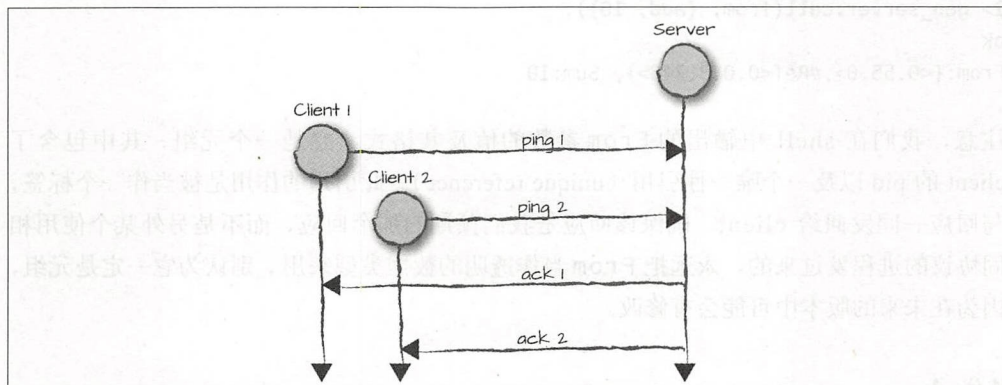


图 4-5：一个 `gen_server` 消息会和（rendezvous）问题

注1 如果你在 `shell` 里运行这个例子，会看到错误报告，原因是 `exit` 信号通过链接传播给了 `shell`，导致了 `shell` 进程的终止。

这个问题的解决办法很简单。你还记得 `handle_call(Message, From, State)` 回调函数里的 `From` 字段吗？我们可以不返回一个 `reply` 给 `behavior` 循环，而是返回 `{noreply, NewState}`。然后使用 `From` 属性以及此函数：

```
gen_server:reply(From, Reply)
```

来延迟对 `client` 的回应，直到我们认为时机成熟。像这种必须同步两个 `client` 的场景，可以在第一次 `handle_call/3` 回调中把 `From`（对应第一个 `client`）存到 `NewState` 里、表里或者数据库里，然后在第二次 `handle_call/3` 回调中再取出。

如果某个同步请求将触发一个耗时的计算，而 `client` 渴望的回应仅仅是一个对请求收到进程已开始执行操作的确认，那么你可以使用 `reply/2` 回应 `client` 而不必等到整个计算全部完成。为了能立刻发送确认，可以在回调中调用 `gen_server:reply/2`：

```
handle_call({add, Data}, From, Sum) ->
    gen_server:reply(From, ok),
    timer:sleep(1000),
    NewSum = add(Data, Sum),
    io:format("From:~p, Sum:~p~n", [From, NewSum]),
    {noreply, NewSum}.
```

运行上述代码，并假定这段代码位于某个 `gen_server` 的回调模块中。其中调用的 `timer:sleep/1` 会暂停进程，允许 `shell` 进程在 `io:format/2` 被调用前收到并处理来自 `gen_server:reply/2` 的回应：

```
1> gen_server:start({local, from}, from, 0, []).
{ok,<0.53.0>}
2> gen_server:call(from, {add, 10}).
ok
From:{<0.55.0>,#Ref<0.0.3.248>}, Sum:10
```

注意，我们在 `shell` 中输出的 `From` 参数的值及其格式。它是一个元组，其中包含了 `client` 的 `pid` 以及一个唯一性引用（unique reference）。此引用的作用是被当作一个标签，与回应一同发回给 `client`，确保该回应是我们预期的那个回应，而不是另外某个使用相同协议的进程发过来的。永远把 `From` 当作透明的数据类型来用；别认为它一定是元组，因为在未来的版本中可能会有修改。

终止

当我们想停止某个 `gen_server` 时该怎么做呢？到目前为止，我们已经见过回调函数 `init/1`、`handle_call/3`、`handle_cast/2` 各自返回的 `{ok, LoopData}`、`{reply, Reply,`

LoopData} 和 {noreply, LoopData} 了。而停止 server 则要求这些回调返回另一种不同的元组：

- init/1 可以返回 {stop, Reason}
- handle_call/3 可以返回 {stop, Reason, Reply, LoopData}
- handle_cast/2 可以返回 {stop, Reason, LoopData}
- handle_info/2 可以返回 {stop, Reason, LoopData}

这些返回值都会以相同的方式终止进程，就好像调用了 `exit(Reason)` 一样。对于 `call` 和 `cast` 的情形，在退出前，回调函数 `terminate(Reason, LoopData)` 会被调用。这使得 server 在关闭前有机会清理自身。`terminate/2` 返回的任何值都会被忽略。对于 `init` 的情况，如果在初始化状态的过程中某些部分失败了，应当返回 `stop`。结果是 `terminate/2` 不会被调用。如果我们在 `init/1` 回调中返回 {stop, Reason}，那么 `start_link` 函数返回 {error, Reason}。

◀ 90

在我们的频率服务器例子里，`stop/0` client 函数会发送一个异步式消息给 server。一旦收到，`handle_cast/2` 回调就会返回带有 `stop` 控制原子的元组，进而导致发起 `terminate/2` 调用。看看下面的代码：

```
stop() -> gen_server:cast(frequency, stop). % frequency.erl

handle_cast(stop, LoopData) ->
    {stop, normal, LoopData}.

terminate(_Reason, _LoopData) ->
    ok.
```

为使例子简单，我们把 `terminate` 留空了。在理想的世界里，我们也许能够在此时杀死所有分配到频率的 `client` 进程，进而使得那些使用这些频率的任务也终止，确保下一次重启时，所有频率都处于可用状态。

注意看 `gen_server:cast/2` 发送给频率服务器的消息。你会注意到它是原子 `stop`，在 `handle_cast/2` 调用中作为第一个参数被模式匹配。这条消息并没有什么特别的含义，只不过是我们在自己代码里这么写罢了。其实发送任何其他的原子都没问题，比如 `gen_server:cast(frequency, donald_duck)`。然后修改 `handle_cast/2` 在模式上匹配 `donald_duck`，最终的结果完全相同。唯一具有特殊含义的是那个出现在 `handle_cast/2` 返回的元组中的第一个元素 `stop`，因为它会被 `gen_server` 的“接收 - 求值”循环进行特别的解释。

如果你关闭 server 是源于正常的工作流需求（例如，该 server 处理的套接字已关闭，或者它控制并监视的硬件已经关闭），那么实践中比较好的做法是把关闭理由 `Reason` 设置



为 normal。而非 normal 的关闭理由尽管也能毫无问题地被接受，但会导致 SASL logger 记录下一条错误报告日志。这样的条目可能会掩盖真正的崩溃情况。（SASL logger 是你使用 OTP 时的另一个赠品。我们会在第 9 章介绍它。）

尽管 server 可以通过返回 stop 元组来正常停止，但也可能由于出现运行时错误而终止的情况。在这类情况下，如果 gen_server 是设定为捕捉退出信号的（通过调用 process_flag(trap_exit, true) BIF），则 terminate/2 也还是会被调用，如图 4-6 所示。如果没有设定为捕捉退出信号，则进程将直接终止，不会调用 terminate/2。

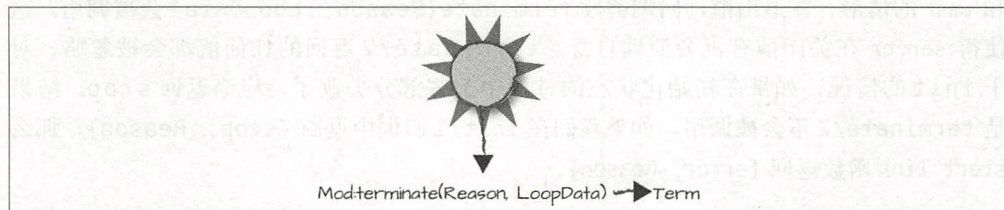


图 4-6: server 非正常终止

91 如果你希望 terminate/2 函数在进程非正常终止后执行，则必须设定 trap_exit 标志。如果没有设置，supervisor 或者相链接的其他进程将会使得 server 直接关掉而没有机会清理自身。

话已至此，请务必确保检查终止时的上下文情况。如果运行时错误发生了，一定要极其仔细地清理好 server 状态，否则可能导致数据受损，而后当 server 重启时引发更多的运行时错误。当重启时，你应该从正确（而且唯一）的数据来源处重建 server 状态，而不是使用上一次崩溃前复制的数据，因为这种数据可能已经受损，进而再次导致崩溃。

调用超时

当使用 gen_server 调用向你的 server 发送同步式消息时，你可以预期回应会在几毫秒后收到。但是假如发送回应出现了延迟会怎样？你的 server 可能正在极其忙碌地处理成千上万的请求，或者其依赖的外部资源存在瓶颈，如数据库、认证服务器、IP 网络或者任何其他消耗一定时间才能回应的资源或 API。OTP behavior 在同步式 API gen_server:call 中内置了 5 秒超时机制。这一时长对于一个软实时系统中的绝大多数查询应该都是足够的了，但也有一些特殊情况需要单独处理。如果你正在使用 OTP behavior 发送一个同步式请求，并且没有在 5 秒内收到回应，那么 client 进程将会触发一个异常。让我们在 shell 里用下面的回调模块试试看：




```
-module(timeout).
-behavior(gen_server).

-export([init/1, handle_call/3]).
```

```
init(_Args) ->
    {ok, undefined}.
```

```
handle_call({sleep, Ms}, _From, LoopData) ->
    timer:sleep(Ms),
    {reply, ok, LoopData}.
```

在 `gen_server:call/2` 函数里，我们发送一条格式为 `{sleep, Ms}` 的消息，其中 `Ms` 是一个值，`handle_call/3` 回调会使用它来调用 `timer:sleep/1`。发送一个超过 5000 毫秒的值会导致 `gen_server:call/2` 函数抛出一个异常，因为这么大的一个值已经超出了默认的超时值。让我们在 shell 里试试看。假设超时模块已经编译好，为的是避免编译器警告我们省略了一些回调函数：

◀ 92

```
1> gen_server:start_link({local, timeout}, timeout, [], []).
{ok,<0.66.0>}
2> gen_server:call(timeout, {sleep, 1000}).
ok
3> catch gen_server:call(timeout, {sleep, 5001}).
{'EXIT',{timeout,{gen_server,call,[timeout,{sleep,5001}]}}}
4> flush().
Shell got {#Ref<0.0.0.300>,ok}
5> gen_server:call(timeout, {sleep, 5001}).
** exception exit: {timeout,{gen_server,call,[timeout,{sleep,5001}]}}
    in function gen_server:call/2
6> catch gen_server:call(timeout, {sleep, 1000}).
{'EXIT',{noproc,{gen_server,call,[timeout,{sleep,1000}]}}}
```

我们启动了 server，并且在 shell 命令 2 中发送了一条同步式消息，告诉 server 睡眠 1000 毫秒后再回应我们原子 `ok`。因为这尚处于 5 秒默认超时内，所以我们成功获得了回应。但是在 shell 命令 3 中，我们把时间提高到 5001 毫秒，导致 `gen_server:call/2` 函数抛出异常。在我们的例子里，shell 命令 3 捕获了异常，这样 client 中的函数有机会处理各种可能触发超时的特殊情形。

如果你决定捕获由于超时而产生的异常，我必须提醒你一句：如果 server 存活但是很繁忙，它将会在超时异常触发后发送回应，而这一回应必须被处理。如果 client 自身是 OTP behavior，此种情况将导致 `handle_info/2` 被调用。如果没有正确实现，client 进程将崩溃。



如果调用来源于纯 Erlang client, 该回应将会被存储在 client 的邮箱里并且永远不会被处理。在邮箱里留存不读取的消息将会消耗内存, 并在进程后续接收消息过程中减慢速度, 原因是模式匹配时这类垃圾消息也会被遍历。不仅如此, 发送消息给一个拥有大量未读消息的进程还会降低发送者的速度, 因为发送操作将消耗更多的余量。这将导致连锁效应, 可能触发更多的超时以及更进一步增加 client 邮箱中垃圾消息的数量。

发送消息给某个拥有很长消息队列的进程时会引发性能惩罚 (performance penalty), 但有一种情形例外, 那就是当 behavior 以同步方式回应发起请求的 client 进程时。如果 client 进程有长消息队列, 感谢编译器和虚拟机优化, receive 分句匹配回应时并不需要遍历整个消息队列。

我们在 shell 命令 4 中看到了这种内存泄漏的证明, 我们冲刷掉了其中的未读消息。如果没有冲刷, 则这条消息将会一直保留在 shell 的邮箱里。在本书里, 我们一直不断地提醒你不要在自己的代码里处理各种边界情况以及意外错误, 因为你引入的故障与错误很可能比你实际解决的还要多。这是一个典型的例子, 这些超时导致的副作用只有当真实系统处于极高负载时才会显现。

现在来看看 shell 命令 5 和图 4-7。我们有一个调用导致了 client 进程崩溃, 因为这个调用执行时并没有在任何 try-catch 语句内。在大多数情况下, 如果你的 server 因为某种 (也许未知的) 原因而不回应了, 则让 client 进程终止并由 supervisor 来处理可能是最好的办法。在这个例子里, shell 进程终止了, 并且马上重启了。超时 server 在 5001 毫秒后发送了一个回应到老的 client (shell) pid。因为这个进程已经不复存在, 因此消息被丢弃了。那为什么 shell 命令 6 失败了且原因是 noproc 呢? 再观察一下 shell 命令序列, 看看你能否再继续阅读前找出答案。

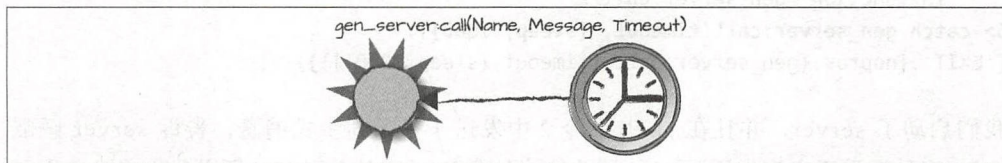


图 4-7: server 超时

当启动 server 后, 我们使其链接到了 shell, 这样 shell 进程相对于 server 进程既是 client 同时也是 parent。在 shell 命令 5 处, 我们执行了一个不带 try-catch 的 gen_server:call/2, 然后超时 server 终止了。由于 server 并未设定为捕捉退出信号, 因此当 shell 终止后, EXIT 信号传播给了 server, 导致它也终止。在正常情况下, server 的 client 和 parent (指与 server 相链接的进程) 应当不是同一个进程, 因此这种情况不会发生。这些问题发生于我们从 shell 测试 behavior 时, 因此你自己练习时请留意。



好吧，我们不想用 behavior 默认的 5 秒而是其他值时该怎么做呢？很简单：填写出我们想要的超时值即可。在 `gen_server` 里，通过使用下述函数调用来实现：

```
gen_server:call(Server, Message, Timeout) -> Reply
```

其中 `Timeout` 是我们想要的毫秒数，或者是原子 `infinity`。

一次 client 调用背后常常连锁触发一系列针对不同 behavior 进程的同步式请求，而且这些 behavior 可能是分布式的。它们各自可能又会向外部资源发出请求。大部分时候，如何选择合适的超时值是一个棘手的问题，因为这些进程访问的服务和 API 是由第三方提供的，完全不在你的控制之内。某些绝大多数情况下可以在若干毫秒内做出回应的系统，当遇到极端负载时可能需要数秒甚至数分钟才能回应。你的系统每秒的吞吐量可能还是一样的，但当有更高——高很多个数量级——的负载通过时，每个请求的延迟将会变高。

◀ 94

要回答 `Timeout` 应当设定为什么值，唯一合适的办法就是以外部需求带动来分析。如果 client 指定了 30 秒的超时，就以此为基准设计后续的请求链。你依赖的外部资源承诺的回应时长为多少？当在极端负载情况下，磁盘访问和 I/O 响应将会如何变化？网络延迟呢？花费一些时间在目标硬件上测试你的系统然后进行相应的调优。当你心里不确定时，就用 5000 毫秒作为默认值开始。使用 `infinity` 时要极其小心，避免使用它除非你别无选择。

死锁

想象某个设计很糟糕的系统中有这样两个 `gen_server`。server1 向 server2 发起了一个同步式调用。server2 收到请求后，经过一系列对其他模块的调用，最终在某处（可能不知情的情况下）执行了一个对 server1 的同步式调用。观察图 4-8，解决问题的方法不是靠什么复杂的死锁预防算法，而是借助超时。

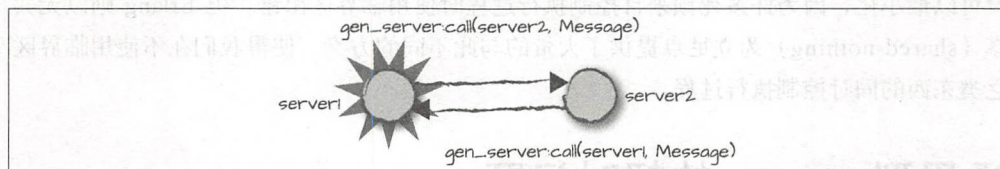


图 4-8: `gen_server` 死锁

如果 server1 在 5000 毫秒内没有收到回应，它将终止，导致 server2 也终止。根据先后顺序不同，触发终止的要么是监视器信号，要么是自身超时。如果还有其他进程也卷入了死锁，终止也会传向它们。supervisor 将会接收到 `EXIT` 信号，并因此而重启所有 server。终止发生于何处这一信息将会被存储到日志文件中，之后我们可以修复引发死锁的故障原因。



避免死锁的策略

构造死锁是很简单的，实际上它们出现的概率非常低，哪怕你的程序极其复杂。这与系统架构、并发模型、进程/应用间的依赖关系，以及处理方式有关。由于 Erlang 不存在共享内存以及临界区，死锁的危险性被大大降低。有经验的 Erlang 程序员在设计程序时默认就会避免在其中出现死锁，因而后续也就不需要操心这个问题。然而新手就不同了，他们应当在系统设计的早期就借鉴并遵循一些适当的策略。对于静态进程场景——也就是说不存在动态启动/终止的进程的场景，一种标准的实践经验是——只允许向前辈进程发起同步式调用，所谓前辈进程，也就是比发起调用的（晚辈）进程启动得更早的那些进程。而反过来，前辈进程如果要调用晚辈进程，则只能是异步式调用。如果前辈进程异步式调用晚辈进程后需要获得回应的话，那么回应需要通过某个（可能是异步的）回调函数返回。静态进程的启动次序是定义在监督树中的，这样的次序关系同样也适用于动态进程。我们在第 8 章介绍进程的重启次序（restart order）后，这一点会变得更清晰。你需要牢记这些知识，因为当你把进程组合为监督树，然后把监督树组合为应用，并再进一步定义应用的启动顺序时，这些知识都会很有用。

我在工作中使用 Erlang 有 17 年了，只遇到过一次死锁。^{注 1} 进程 A 同步式调用进程 B，而 B 又发起了向另一个节点的远程过程调用（remote procedure call），而那个节点同步式调用了进程 C。进程 C 同步式调用了进程 D，进程 D 则发起了向最初第一个节点的远程过程调用。这个调用导致了一个对进程 A 的同步式调用，而此时的进程 A 还正等着进程 B 的回应呢。这个死锁是在我们首次尝试将两个节点整合时发现的，然后为了解决它我们只花了 5 分钟。进程 A 应当异步式调用进程 B，然后进程 B 使用一个异步式回调来把结果发回给 A。所以，死锁的风险确实存在，但如果你采用正确的方法面对，这种风险可以最小化，因为许多死锁来自控制执行过程时使用临界区出错，但 Erlang 则以无共享（shared-nothing）为立足点提供了大量的与此不同的方案，使得我们在不使用临界区之类东西的同时控制执行过程。

通用型 server 的超时问题

想象有这么一个 `gen_server`，它的任务是监视某种特别的硬件设备并与其通信。如果超过一个预定的时间 server 没收到来自该设备的消息，则 server 应当主动发送一个 ping 请求来看看该设备是否还活着。这些 ping 请求可以通过内部超时机制来触发，创建方法是在 behavior 回调函数返回的结果的控制元组中增加一个超时值：

注 1 我就是上一本书里那个导致全国移动网络数据中断的人。




```

init/1      -> {ok, LoopData, Timeout}
handle_call/3 -> {reply, Reply, LoopData, Timeout}
handle_cast/2 -> {noreply, LoopData, Timeout}
handle_info/2 -> {noreply, LoopData, Timeout}

```

值 Timeout 要么是一个整数，代表毫秒级的时间，要么是原子 infinity。如果 server 在 Timeout 设定的毫秒时间内没有收到一条消息，那么它将会在其 handle_info/2 回调函数中收到一条 timeout 消息。返回 infinity 则相当于没有设置超时值。让我们通过一个小例子来试试吧，每隔 5000 毫秒产生一次超时，然后获取当前时间并输出秒数。我们可以通过发送同步式消息 start 和 pause 来暂停和重启计时器：

```

-module(ping).
-behavior(gen_server).

-export([init/1, handle_call/3, handle_info/2]).
-define(TIMEOUT, 5000).

```

```

init(_Args) ->
    {ok, undefined, ?TIMEOUT}.

handle_call(start, _From, LoopData) ->
    {reply, started, LoopData, ?TIMEOUT};
handle_call(pause, _From, LoopData) ->
    {reply, paused, LoopData}.

```

```

handle_info(timeout, LoopData) ->
    {_Hour, _Min, Sec} = time(),
    io:format("~2.w~n", [Sec]),
    {noreply, LoopData, ?TIMEOUT}.

```

假定此 ping 模块已编译，我们启动它并每隔 5 秒产生一次超时。可以通过发送 pause 消息来暂停超时，原因是当 server 收到这一消息后会通过 handle_call/3 的第二个分句处理，而该分句返回的元组里并未包含 timeout。然后我们又可以发送 start 消息来把它重新开启：

```

1> gen_server:start({local, ping}, ping, [], []).
{ok,<0.38.0>}
22
27
2> gen_server:call(ping, pause).
paused
3> gen_server:call(ping, start).
started

```



51

56

4> gen_server:call(ping, start).

started

4

因为我们设定的超时时间相对较长，因此并不会真的每隔 5000 毫秒就生成一条超时消息，而是只有在 behavior 在整个超时时长范围内都没有收到任何消息时才会。如果收到了任何消息，就像我们在 shell 命令 4 中所做的那样，将会使得超时计时器被重置。

如果你需要一个不会被重置或者必须以固定间隔触发，不受收到消息与否影响的定时器，那么请使用 `erlang:send_after/3` 或 `timer` 模块提供的一些函数，包括 `apply_after/3`、`send_after/2`、`apply_interval/4` 以及 `send_interval/2`。

使 behavior 休眠

如果我们返回的超时值不是原子 `infinity` 而是原子 `hibernate`，则 `server` 将会降低其内存消耗进入到等待状态。当 `server` 属于间歇性收到请求，而这些请求触发的又是消耗内存的操作，会导致系统运行的内存不足时，你就会想要使用这一 `hibernate` 功能。使用 `hibernate` 将会丢弃调用栈，并运行一次完整的垃圾回收，把一切归置到一块连续的堆中。堆中已分配的内存尺寸将会收缩到与数据所需相同。`server` 将保持此状态直到收到一条新消息。



通过这种方式休眠进程是有代价的，因为它涉及在休眠前和唤醒后进行 `full-sweep` 垃圾回收。除非你能预见到在未来一段时间内 `behavior` 都不会收到消息，并且你有节约内存的需要，才使用休眠机制，反之如果 `server` 在短时间会蜂拥收到一系列消息的情况下不要使用休眠机制。把休眠当作一种抢救 (`preemptive`) 手段是危险的，特别是如果你的进程是繁忙的，这种操作可能会（而且是极可能会）花费更多的资源去完成休眠，多到比放任不管还多。唯一能够确定是否应该使用休眠的办法是对系统在压力环境下进行基准测试，证明当降低内存占用时会获得性能增益。请三思而后行。如有疑问，别去做！

全局化

`behavior` 进程可以注册到本地，也可以注册到全局。在我们的例子里，都是注册到本地，使用的是格式为 `{local, ServerName}` 的元组，其中 `ServerName` 是一个原子，表示别名。这种做法等价于使用 `register(ServerName, Pid)` BIF 来注册进程。如果想在分布式集群环境下实现位置透明性该怎么办呢？

98 全局注册的进程由全局名字服务器 (global name server) 承载，这样一来，在分布式节



点（可能分过区的）集群里便能透明地访问它们。名字服务器会把名字清单复制给全部节点，并监视每个节点的健康状况和网络连通性，确保没有中央点故障（central point of failure）。在服务进程名参数处使用 `{global, Name}` 元组就可以把服务进程注册到全局。这样做等价于使用 `global:register_name(Name, Pid)` 函数来注册。接下来，在你发起同步或异步调用时，也使用这一元组：

```
gen_server:start_link({global,Name},Mod,Args,Opts) ->
    {ok, Pid} | ignore | {error, Reason}
gen_server:call({global, Name}, Message) -> Reply
gen_server:cast({global, Name}, Message) -> ok
```

如果你想把全局进程注册的内部实现机制替换为自己的方案，有一个 API 允许你这么做。当你觉得 `global` 模块提供的功能不足，或者当你想根据网络拓扑不同而有针对性地使用不同的 `behavior` 时，可以借助它创建自己的实现。你需要提供一个回调模块——在这里我们称之为 `Module`——其需要导出与 `global` 模块相同的那些函数，并且返回值也相同，包括 `register_name/2`、`unregister_name/1`、`whereis_name/1` 以及 `send/2`。然后注册名字时使用元组 `{via, Module, Name}`，并且启动你的进程时使用 `{via, global, Name}`，与使用 `{global, Name}` 全局注册时一样。对于全局注册的进程，`Name` 并非只能是原子，实际上任何 Erlang 数据项都是可以的。一旦把这个回调模块准备好了，你就可以这样启动你的进程并发送消息：

```
gen_server:start_link({via, Module, Name},Mod,Args,Opts) -> {ok, Pid}
gen_server:call({via, Module, Name}, Message) -> Reply
gen_server:cast({via, Module, Name}, Message) -> ok
```

在本书余下的部分，我们使用 `NameScope` 统称 `{via, Module, Name}`、`{local, Name}` 和 `{global, Name}`。大部分的 `server` 都是注册在本地的，但是根据系统复杂度和集群策略不同，也会使用全局注册以及自定义全局注册。

当与 `behavior` 通信时，你可以不使用它们注册的别名而直接使用它们的 `pid`。注册 `behavior` 不属于强制要求；而且不注册 `behavior` 你也可以同时运行同一个 `behavior` 的多个实例。当启动 `behavior` 时，只需省略名字字段：

```
gen_server:start_link(Mod, Args, Opts) ->
    {ok, Pid} | ignore | {error, Reason}
```

如果你想把一个请求广播给一个节点集群内的所有 `server`，可以使用 `gen_server` 提供的 `multi_call/3` 函数，该函数与 `abcast/3` 函数一样都能广播，但 `multi_call/3` 会返回结果，而 `abcast/3` 不会返回。

```
gen_server:multi_call(Nodes, Name, Request [, Timeout]) ->
```



```

{{[{Node,Reply}], BadNodes}
gen_server:abcast(Nodes, Name, Request) -> abcast

```

相应的，每个节点上运行的 `server` 将会通过 `handle_call/3` 和 `handle_cast/2` 回调函数来处理这些请求。当使用 `abcast` 异步地广播时，对于目标节点是否已连接，以及是否存活是不做检查的。无法抵达目标节点的请求，将简单地被抛弃。

链接 behavior

当你在 `shell` 里启动 `behavior` 时，`shell` 进程会和 `behavior` 进程相链接。如果 `shell` 进程非正常终止，它的 `EXIT` 信号将会传播给当初由它启动的那些 `behavior` 进程，进而导致它们也终止。如果通过调用 `gen_server:start/3` 或 `gen_server:start/4` 来启动 `gen_server`，则这些进程就不会与它们的父进程链接。使用这些函数时请小心，尽量只在开发和测试中使用，因为 `behavior` 总是应当与父进程链接的。

```

gen_server:start(NameScope, Mod, Args, Opts)
gen_server:start(Mod, Args, Opts) ->
    {ok, Pid} | {error, {already_started, Pid}}

```

Erlang 系统会持续数年运行于不重启的计算机上。即使为了修复错误，完善功能或者增加新功能而需要软件升级时，或者 `behavior` 非正常终止进行重启时，这些系统也能持续运行。当关闭一个子系统时，你需要百分百地确保与该子系统相关的进程都终止了，避免留下孤儿进程。这就得靠链接机制来实现。第 8 章我们涉及 `supervisor behavior` 的时候会介绍更多细节。

总结

本章里我们介绍了 `gen_server behavior` 最重要的概念和功能，它是其他所有 `behavior` 的基础。现在你应该已经清楚为什么要使用 `gen_server behavior`，这么做相较于自己再造个轮子优势何在。对于使用该 `behavior` 时所需了解的那些函数与回调，我们的介绍已经涵盖了其中的大多数。尽管你并不需要深入了解这些技术幕后的原理，还是希望你能明白事情远比眼见的复杂。我们介绍过的内容里，最重要的那些函数已经列在了表 4-1 里。

100 表 4-1: `gen_server` 的回调函数

gen_server 函数 / 操作	gen_server 回调函数
<code>gen_server:start/3</code> , <code>gen_server:start/4</code> , <code>gen_server:start_link/3</code> , <code>gen_server:start_link/4</code>	<code>Module:init/1</code>



gen_server 函数 / 操作	gen_server 回调函数
gen_server:call/2, gen_server:call/3, gen_server:multi_call/2, gen_server:multi_call/3	Module:handle_call/3
gen_server:cast/2, gen_server:abcast/2, gen_server:abcast/3	Module:handle_cast/2
各种消息, 包括来源于 Pid ! Msg 操作的, 或者是来自 process/node monitor, 或者是端口以及套接字的, 以及 exit 息和其他非 OTP 类消息	Module:handle_info/2
通过返回 {stop, ...} 而触发, 或是在捕捉 exit 状态下非正常终止而触发的	Module:terminate/2

当编译 behavior 模块时, 你会看到一条警告信息, 提示 code_change/3 回调找不到。在第 11 章讲到发行包制作以及软件升级的时候, 我们会谈到它。在下一章, 在使用 gen_server behavior 当作例子的时候, 我们会涉及一些高级主题, 以及随 OTP 自带的一些 behavior 特定的功能。

此刻, 你或许应该好好温习一下 gen_server 模块的说明手册页面。如果你感觉自己很勇敢, 那就读读 gen_server.erl 源代码文件, 以及 gen 助手模块的源码。读完本章, 加上对前面章节以及各种边界情形 (corner case) 的理解, 你会发现代码其实也并非像看上去那么晦涩。

接下来是什么

下一章有一些零碎的内容, 让你可以进一步深挖 behavior。首先是一些内置的与追踪和日志相关的功能, 我们将通过学习它们的用法来了解它们。我们还会介绍 start 函数里的 Opts 标志。借助这些标志, 你可以优化性能、调整内存使用, 以及启动你的 behavior 时开启追踪功能。所以继续读吧, 下一章有精彩又有趣的内容。



深入控制OTP行为模式

前一章我们介绍了 `gen_server` 行为模式 (behavior) 的各个主要方面，至此你已经实现了自己的第一个 client-server 应用程序，并且认识到了 OTP 行为模式能够帮助你减少基础代码的编写工作量，使你能够聚焦于系统开发中与当前需求更加紧密相关的部分。本章我们会更进一步深入挖掘行为模式，探索一些更加高级的主题并穿插介绍相关的一些内置功能。虽然我们目前围绕 `gen_server` 进行探讨，但是我们写的绝大多数代码也适用于许多其他的 behavior，包括你自己实现的那些。仔细阅读，因为本章的内容会被后续章节频繁引用到。

sys 模块

我们曾经多次提到，使用 OTP 行为模式时会获得许多内置功能，还说过可以在此基础上添加自己的新功能。其中我们提过的大部分功能都是通过访问 `sys` 模块获得的，它让我们可以生成追踪事件、探查并操作行为模式状态，以及收发系统消息。这些功能适用于标准 OTP 行为模式，但是就如我们在第 10 章展示的那样，你也可以在定义自己的行为模式时使用到它们。

追踪与记录

我们通过运行一个小例子来了解内置的追踪机制是如何运作的。在 `shell` 里启动你的频率服务器，并使用 `sys` 模块，按照如下步骤进行操作：

```
1> frequency:start().
{ok,<0.35.0>}
2> sys:trace(frequency, true)..
ok
3> frequency:allocate().
```



```

*DBG* frequency got call {allocate,<0.33.0>} from <0.33.0>
*DBG* frequency sent {ok,10} to <0.33.0>,
    new state {[11,12,13,14,15],[{10,<0.33.0>}]}
{ok,10}
4> frequency:deallocate(10).
*DBG* frequency got cast {deallocate,10}
ok
*DBG* frequency new state {[10,11,12,13,14,15],[]}
5> sys:trace(frequency, false).
ok

```

通过为 frequency 分配器开启追踪标志 (trace flags)，我们能够针对系统事件——包括消息收发、状态改变等——生成输出。在我们的例子中，消息被管道输出至 shell。如果我们改用 `sys:log/2` 调用，则这些消息将被存储到 server 的循环过程中。可以使用 `print` 标志把它们显示出来，或者使用 `get` 标志以 Erlang 数据结构的方式获取：

```

6> sys:log(frequency, true).
ok
7> {ok, Freq} = frequency:allocate(), frequency:deallocate(Freq).
ok
8> sys:log(frequency, print).
*DBG* frequency got call {allocate,<0.33.0>} from <0.33.0>
*DBG* frequency sent {ok,10} to <0.33.0>,
    new state {[11,12,13,14,15],[{10,<0.33.0>}]}
*DBG* frequency got cast {deallocate,10}
*DBG* frequency new state {[10,11,12,13,14,15],[]}
ok
9> sys:log(frequency, get).
{ok,[{in,{ '$gen_call'},{<0.33.0>,#Ref<0.0.4.59>},
    {allocate,<0.33.0>}},
    frequency,#Fun<gen_server.0.40920150>},
{{out,{ok,10},<0.33.0>,[{11,12,13,14,15],[{10,<0.33.0>}]}},
    frequency,#Fun<gen_server.6.40920150>},
{{in,{ '$gen_cast'},{deallocate,10}}},
    frequency,#Fun<gen_server.0.40920150>},
{{noreply,[{10,11,12,13,14,15},[]]}},
    frequency,#Fun<gen_server.4.40920150>}}]}
10> sys:log(frequency, false).
ok

```

当使用 `sys:log/2` 调用把追踪事件存储到 server 循环中时，默认存储的事件数是 10。你可以在启用记录时通过传入 `{true, Int}` 标志来改写这一数字。Int 为一个整数，代表你想设定的默认的事件存储数量。当你计划处理大量的调试消息，或者打算长时间开启此调试功能时，使用 `sys:log_to_file/2` 来把消息管道输出到文本文件中去。

系统消息

看前一个例子里 shell 命令 9 的返回值。如果传 `get` 标志给 `sys:log/2`，我们会得到系统事件的一个列表。日志中的事件的格式与产生它们的进程有关，但一般而言，每个事件都包含如下形式的系统消息：

```
{in, Msg}
```

此消息的产生源于有一条消息（包括超时）发往了 `gen_server`。Msg 包括任何符合 OTP 消息协议的结构，例如针对 `cast` 的 `{'$gen_cast', Msg}`，以及针对 `call` 的 `{'$gen_call', {Pid, Ref}, Msg}`。对于任何以消息形式发往 `gen_server` 进程的 Erlang 数据项，Msg 将被简单地设为相同的值。

```
{out, Msg, To, State}
```

当使用 `{reply, Reply, NewState}` 控制元组回应 client 时将生成此系统消息，但如果使用的是 `gen_server:reply/2` 则不会生成此系统消息。Msg 是发给 client 的回应内容，而 To 则是 client 的 pid。State 与回应元组中的 `NewState` 相同。

```
term()
```

任何其他格式的系统消息也是允许的。举一个例子，shell 命令 9 的返回值包含消息 `{noreply, {[10,11,12,13,14,15],[]}}`，它是 `handle_cast/2` 处理 `deallocate cast` 后的结果。noreply 元组中的第二个元素是 `gen_server` 的新状态。

注意，到本文编写时（包括 Erlang 18 在内），sys 模块也把 `{in, Msg, From}` 和 `{out, Msg, To}` 视为合法的系统消息，但是它们没有被其他任何标准行为模式使用。

你自己的追踪函数

你可以实现自己的追踪函数（trace functions），办法是实现一些 fun（函数），它们会配合系统事件而被触发。你可以对这些事件进行模式匹配，然后做任何你想做的事。追踪函数可以用作多种用途，包括生成你想要的调试输出、使用 `dbg` 或者追踪 BIF 开启低级追踪、启用特定信息的记录操作、运行诊断函数，或者执行任何你可能需要的代码（或者什么也不执行）。

104 下面的例子使用了一个计数器来记录 client 请求分配 frequency 被拒绝的次数，并每次输出一条警告消息。^{注1} 注意，我们是如何在不更改原始 frequency 代码的情况下达成这一目的的：

注1 此函数中执行的 `io:format/2` 会把自身附着（attach）到被追踪的行为模式的分组领导者（group leader）上，导致本地 shell 中输出了警告提示。但如果你是通过远程 shell 连接的话，则不会看到这些信息。


```

11> F = fun(Count,{out, {error, no_frequency}}, Pid, _LoopData}, ProcData) ->
    io:format("*DBG* Warning, Client ~p refused frequency! Count:~w~n",
    [Pid, Count]), Count + 1;
    (Count, _, _) ->
        Count
    end.
#Fun<erl_eval.18.54118792>
12> sys:install(frequency, {F, 1}).
ok
13> frequency:allocate(), frequency:allocate(), frequency:allocate(),
    frequency:allocate(), frequency:allocate(), frequency:allocate().
{ok,15}
14> frequency:allocate().
*DBG* Warning, Client <0.33.0> refused frequency! Count:1
{error,no_frequency}
15> frequency:allocate().
*DBG* Warning, Client <0.33.0> refused frequency! Count:2
{error,no_frequency}
16> sys:remove(frequency, F).
false
17> frequency:allocate().
{error,no_frequency}

```

让我们把这个例子看得更仔细一些。我们创建了一个函数 F，其接收三个参数。第一个，Count，是此调试函数的状态，在各次调用中传入。Count 在这个例子里扮演了状态变量的角色，因为我们选了它来累计第一个函数分句被匹配的次数。其他追踪函数可能会使用更复杂的状态。第二个参数则是系统消息，我们针对格式如 {error, no_frequency} 的出站 (outbound) 消息做了模式匹配。第三个参数，ProcData，则取决于具体追踪的是什么样的行为模式；举一个例子，对于一个 gen_server，它是一个进程已注册的名称或 pid，而对一个 gen_fsm，它则是一个元组，包含了进程名称或 pid 以及该 FSM（有限状态机）当前的状态名（我们将在第 6 章介绍 gen_fsm）。F 函数的第二个分句则起到忽略其他一切系统消息的作用。我们在 shell 命令 12 中，sys:install/2 调用的时候，通过元组中的第二个元素把调试函数的状态 Count 设置为整数 1。在这条命令里，我们同时还把 F 传给了频率服务器，开启了调试输出。通过调用 frequency:allocate/0 足够多次耗尽了可用频率，两次触发了调试输出并使计数增加了。每次该函数执行，F 如果匹配的是第一个分句，则 Count 增加 1，否则 Count 保持不变。在调试函数里返回原子 done 后调试函数就被禁用，这样做与直接调用 sys:remove/2 等价，与 shell 命令 16 相同。

◀ 105

统计信息和当前状态

sys 模块还允许你收集行为模式的综合统计信息以及内部状态信息，包括循环数据，而这些都是不需要你重新发明轮子或者实现任何新东西。

```

18> sys:statistics(frequency, true).
ok
19> frequency:allocate().
{error,no_frequency}
20> sys:statistics(frequency,get).
{ok,[{start_time,{2015,11,29},{20,10,54}}},
    {current_time,{2015,11,29},{20,12,9}}},
    {reductions,33},
    {messages_in,1},
    {messages_out,0}]}
21> sys:statistics(frequency, false).
ok
22> sys:get_status(frequency).
{status,<0.35.0>,
 {module,gen_server},
 [[{'$ancestors',[<0.33.0>]],
  {'$initial_call',{frequency,init,1}}],
 running,<0.33.0>,[],
 [{header,"Status for generic server frequency",
  {data,[{"Status",running},
        {"Parent",<0.33.0>},
        {"Logged events",[]}]},
  {data,[{"State",
        {{available,[],
          allocated,[{15,<0.33.0>},
                     {14,<0.33.0>},
                     {13,<0.33.0>},
                     {12,<0.33.0>},
                     {11,<0.33.0>},
                     {10,<0.33.0>}]}}]}]}]}]}]}

```

`sys:statistics/2` 返回了一个列表，其中的值带有自解释的标签，而 `sys:get_status/1` 返回的元组就不那么直观，它返回一个如下格式的元组：

```
{status, Pid, {module,Mod}, [ProcessDictionary, SysState, Parent, Dbg, Misc]}
```

其中 `Pid` 和 `Mod` 分别对应行为模式的进程标识符和回调模块。`ProcessDictionary` 是一个由 `key-value` 元组构成的列表。注意，尽管我们没有在频率服务器的例子里用到进程字典（process dictionary），但是 `gen_server` 库模块以及其他我们还没有提到的行为模式都用到了这一功能。

106 `SysState` 告诉我们行为模式的状态是处于 `running`（运行）还是 `suspended`（暂停）。通过调用 `sys:suspend/1` 和 `sys:resume/1`，我们可以让行为模式停止处理普通消息（normal messages）的状态，在这种状态下，进程将只处理系统消息（system messages）。一般来说，当你使用 OTP 特有的升级功能升级软件时，或者当测试某些边界条件时你会暂停进

程。还有当定义自己的行为模式时，你也会暂停进程，但是当使用的是标准行为模式时一般不会这么做。在你自己程序的业务逻辑中，要暂停进程只应当通过在邮箱中没有能匹配的项时使用 `receive` 分句实现。在你的代码中使用 `sys:suspend/1` 是错的！

`Parent` 是父进程的 `pid`，会被那些设定为捕捉退出信号的行为模式进程用到。如果父进程终止，当前行为模式进程也不得不终止。在本例中，`Parent` 是 `shell` 进程 ID。`Dbg Flag` 是持有追踪 (`trace`) 和统计 (`statistics`) 方面的标志，在我们获取状态的这一时刻它们都被关闭了 (因此是一个空列表)。

最后，`Misc` 是一个由标签式元组 (`tagged tuples`) 构成的列表，其中包含与行为模式相关的信息。其中的项随行为模式不同而不同，并且，你可以通过在你的行为模式回调模块中提供一个 (可选的) 回调函数，来更改这些信息。当面对的是通用服务器时，`Misc` 中最重要的信息就是循环数据。你可以通过在你的行为模式回调模块中提供一个可选的回调函数来影响 `Misc` 值中的内容，例如使用该函数格式化 `{data, [{"State", ...}]}` 字段为一个对最终用户 (`end user`) 更简单、更有意义或者更有帮助性的形式：

```
...
-export([format_status/2]).
...
```

```
format_status(Opt, [ProcDict, {Available, Allocated}]) ->
    {data, [{"State", {{available, Available}, {allocated, Allocated}}}]}
```

如果 `Opt` 是原子 `normal`，它告诉我们状态是通过 `sys:get_status/1` 调用获取的。如果行为模式是非正常终止的，并且错误报告中包含了其状态值，则 `Opt` 会被设定为 `terminate`。

`ProcDict` 是一个 `key-value` 元组构成的列表，其中包含了进程字典 (`process dictionary`)。在更早的例子中，新状态为：

```
{data,[{"State", {{available, []},{allocated, [{15,<0.33.0>}, {14,<0.33.0>},
                                                {13,<0.33.0>}, {12,<0.33.0>},
                                                {11,<0.33.0>}, {10,<0.33.0>}]}}]}
```

虽然并不强制范围必须是形如 `{data, [{"State", State}]}` 的元组，但建议这么做，因为这能够保持与当前做法的一致性。

如果只是想检查行为模式回调模块的进程中存储的循环数据，使用 `sys:get_state/1`：

```
23> {Free, Alloc} = sys:get_state(frequency).
{[],
 [{15,<0.33.0>}, {14,<0.33.0>}, {13,<0.33.0>}, {12,<0.33.0>},
 {11,<0.33.0>}, {10,<0.33.0>}]}
```

这个便捷的方法使得我们避免使用 `sys:get_status/1` 来抽取循环数据，因为通常在 shell 里交互式调试有点难。`sys:get_state/1` 调用只是用于调试，而函数 `sys:replace_state/2` 则允许替换一个正在运行中的行为模式进程的循环状态。举一个例子，想象你正在 shell 里调试，而你想要快速地添加几个频率。你可以通过重新编译代码，然后重启 server——当频率数不多时，这么做还是很简单的，但如果已分配出的频率成千上万，此时你想做一下测试并且还要保留状态，事情就难得多了：

```
24> sys:replace_state(frequency, fun(_) -> {[16,17], Alloc} end).
[[16,17],
 [15,<0.33.0>], {14,<0.33.0>}, {13,<0.33.0>}, {12,<0.33.0>},
 {11,<0.33.0>}, {10,<0.33.0>}]}
25> frequency:allocate().
{ok,16}
```

替换循环数据需要传入一个函数，该函数接收循环数据的当前值，并返回新值。这允许你轻松地只修改复杂循环数据值中所需的部分。在这个例子里，我们把原本为空列表的可用频率列表替换成一个拥有两个新频率的列表，同时保持已分配频率列表不变。`sys:replace_state/2` 函数返回新的循环数据。由于例子中我们增加了可用频率，接下来调用 `frequency:allocate/0`，本来按之前的情况应该返回 `{error, no_frequency}`，现在则返回 `{ok, 16}`。

sys 模块总结

为了总结，让我们再看一看 `sys` 模块中那些我们已经见过的函数。留意我们在函数描述中所标注的 `[,Timeout]`。其含义是，函数调用时这是一个可选的参数，因而该函数的元数是 2 或者 3。因为这些函数只不过是对我们的行为模式的同步调用，所以使用 `Timeout` 允许改写默认的 5 秒超时时间为一个更适合的值。我们已经介绍过的函数如下：

```
sys:trace(Name,TraceFlag [,Timeout]) -> ok

sys:log(Name,LogFlag [,Timeout]) -> ok | {ok, EventList}
sys:log_to_file(Name,FileFlag [,Timeout]) -> ok | {error, open_file}

sys:install(Name,{Func,FuncState} [,Timeout]) -> ok
sys:remove(Name,Func [,Timeout])

108 sys:statistics(Name,Flag [,Timeout]) -> ok | {ok, Statistics}.

sys:get_status(Name [,Timeout]) -> {status, Pid, {module, Mod}, Status}

sys:get_state(Name [,Timeout]) -> State
```



```
sys:replace_state(Name,ReplaceFun [,Timeout]) -> State
```

```
sys:suspend(Name [,Timeout]) -> ok
```

```
sys:resume(Name [,Timeout]) -> ok
```

为了输出追踪事件到 shell, 使用 `trace/2`。当需要记录事件到日志以后再检索, 使用 `log/2`。通过设定 `LogFlag` 为 `true` 或者 `false` 开启或关闭日志记录。默认情况下, 最后 10 条事件被存储下来; 你可以改写这个值, 方法是使用 `{true, Int}` 开启日志记录, 其中 `Int` 是一个非负整数。

通过使用 `print` 和 `get` 标志可以获取事件。当使用 `log_to_file/2` 时, 事件会被记录为文本格式。 `FileFlag` 是一个字符串, 用于指定绝对的或者相对的文件名, 也可以设为原子 `false` 来关闭它。使用 `sys:install/2` 来写入你自己的与系统事件相关的触发器和追踪函数, 而使用 `sys:remove/2` 来恢复。

当使用 `statistics/2` 时, 通过设置 `Flag` 为 `true` 或者 `false` 可以相应地开启、关闭统计信息的收集。使用 `get_state/1` 来检查循环数据, 以及使用 `replace_state/2` 来替换。并且 `get_status/1` 能返回行为模式与内部状态相关的所有可用数据。当调试和排查在线系统 (live systems) 时, `get_state/1`、`replace_state/2` 和 `get_status/1` 函数非常有帮助。

还记得启动 `server` 时作为最后一个参数传入的 `Opts` 吗? 当时我们用了空的列表占位。你可以在启动行为模式时通过使用 `Opts` 字段来开启追踪、日志记录和统计。如果你传入 `[{debug, DbgList}]`, 其中 `DbgList` 包含一个或多个条目——涉及 `trace`、`log`、`statistics` 和 `{log_to_file, FileName}`——这些标志在行为模式进程启动后立刻就会生效。

分裂时的可选项

当启动一个行为模式的时候, 你可以根据性能和内存利用方面的需要改变默认的内存和垃圾回收器设置。你传入的那些设置实际上与 `spawn_opt/4` BIF 能接受的那些是同一批, 但此处作为 `Opts` 参数传入时格式为 `[{spawn_opts, OptsList}]`, 该数组中除了设置, 还有调试选项。

使用分裂选项时要小心! 要确认你遇到的是与内存管理相关的性能问题, 唯一的办法就是对你的系统做性能剖析和基准测试。在这么做的过程里, 你需要理解底层中堆、内存分配以及垃圾回收方面的工作机制。过早地进行优化是万恶之源 (仅次于共享内存及可变状态)。如果你不相信这一点, 你很快会发现尝试优化内存管理通常反而会导致相反的

109

效果，导致你的程序变慢。绝大多数情况下别去做性能调优——与使用 {global, Name} 全局注册时一样。

内存管理与垃圾回收

如果怀疑你的性能问题可以通过内存管理定位，对你的系统做基准测试时改变一下堆和垃圾回收器设置。与内存相关的可改的选项包括：

`min_heap_size`

设定在触发垃圾回收器 (gc) 执行垃圾回收前进程堆能增长到的尺寸上限。但是这个名字 (`min_heap_size`) 是有误导性的，因为事实上这个值代表了堆在触发 gc 前能增长到的最大 (maximum) 尺寸。

`min_bin_vheap_size`

设定本进程在共享二进制堆中允许使用的空间的初始最小值，超出这一值后就会触发针对二进制数据的垃圾回收。

`fullsweep_after`

决定了必须执行多少次分代式垃圾回收后才能执行一次全量垃圾回收。

BEAM 的垃圾回收机制是如何运作的

Erlang 的垃圾回收可以描述为——以技术术语来说——属于一种分进程分代半空间复制型回收器 (per-process generational semispace copying collector)，其使用了 Cheney 的复制回收算法，同时拥有一个全局的大对象 (large object) 空间。用通俗的话来说就是，当一个进程用完了为其在堆中分配的内存时，BEAM 虚拟机就会触发一次垃圾回收，复制所有活的 (依然在使用中的) 数据到新的堆上，释放所有先前持有的空间。

之所以把此垃圾回收器称为“分代式”的，是因为堆中那些在两次清扫中都活着的数据会被从所谓的年轻堆 (young heap) 复制到一个称为老龄堆 (old heap) 的区域。之所以把数据移动到老龄堆是基于这样一个假设，即对于能够活过两次垃圾回收的数据，很可能也会活得更久。垃圾回收器的运作总是以遍历年轻堆上的数据开始，复制活过前一次垃圾回收的数据到老龄堆上，然后创建一个新的年轻堆存储其余的数据：之前年轻堆的整个内存都会被释放。如果对年轻堆进行垃圾回收已经无法释放出足够的内存 (或者缺少足够的内存可供从年轻堆复制过去)，那么将触发一次全量式垃圾回收 (full-sweep garbage collection)。这将检查并释放老龄堆和年轻堆

上所有不再被引用的数据。

如果这样完整清理后还是缺乏足够的内存,则堆尺寸将通过分配内存块的方式增长,增长是按斐波那契递推数列进行的,该数列的基为 12 字和 38 字 (word)。每一次增长是把数列中前两次的值相加再加 1,因此下一次的尺寸将会是 $38+12+1$,即 51 字。这一过程将持续直到 833 026 字后,则改为以当前尺寸的 20% 递增。

当分代式垃圾回收进行若干次后,如果达到了某个预定次数值,则也会触发一次全量式垃圾回收。由于长期存活的进程周期性地地进行一些小活动,它们可能会拥有一个巨大的已分配堆,其中持有不再使用的数据。解决这种问题,可以配置触发全量式回收所需的分代式垃圾回收次数值,或者对进程进行休眠(参见第 4 章“使 behavior 休眠”一节)。

进程用到的数据以及它自身的状态并非都是存储在它自己的堆里的。超过 64 字节的二进制数据会被存储在一个所有进程都共享的二进制堆 (shared binary heap) 中。这些二进制数据是通过引用 (reference) 来访问的,因此如果通过消息传递这些引用,可以在多个进程间共享。由于利用了引用,因此当把大尺寸的二进制数据作为消息传递时,效率也很高,毕竟不需要去复制。每当有对此种二进制数据的引用时,引用计数器就加 1,然后当该引用移除时,引用计数器就减 1。最终计数器减为 0 时,这个二进制数据就可以被垃圾回收了。

每个进程都有一个非全局共享的虚拟二进制堆 (virtual binary heap)。当某个进程超出了其虚拟二进制堆尺寸而需要释放更多空间时就会触发垃圾回收。小于 64 字节的二进制数据原本是存储在普通堆 (normal heap) 上的,但有两种情况会被复制到虚拟二进制堆上,一是当该二进制数据作为消息的一部分发往其他进程的时候;二是当垃圾回收进行的时候。进程和虚拟二进制堆的垃圾回收都是针对各个进程独立完成的,这减少了由于内存管理而导致的中断,同时保证了系统的软实时属性。

在下面的例子里,我们启动了频率服务器并且追踪与垃圾回收有关的事件。我们使用 *dbg* 追踪器来测量进程花费了多少微秒在垃圾回收上。当分配了 5 个频率时,总共花费了 9 微秒 (911 345–911 336):

```
1> dbg:tracer().
{ok,<0.35.0>}
2> {ok, Pid} = frequency:start().
{ok,<0.38.0>}
3> dbg:p(Pid, [garbage_collection, timestamp]).
{ok, [{matched, nonode@nohost, 1}]}
4> frequency:allocate(), frequency:allocate(), frequency:allocate(),
```

```

    frequency:allocate(), frequency:allocate().
{ok,14}
(<0.38.0>) gc_start [{old_heap_block_size,0},
    {heap_block_size,233},
    {mbuf_size,0},
    {recent_size,0},
    {stack_size,12},
    {old_heap_size,0},
    {heap_size,213},
    {bin_vheap_size,0},
    {bin_vheap_block_size,46422},
    {bin_old_vheap_size,0},
    {bin_old_vheap_block_size,46422}] (Timestamp: {1448,829619,911336})
(<0.38.0>) gc_end [{old_heap_block_size,0},
    {heap_block_size,233},
    {mbuf_size,0},
    {recent_size,44},
    {stack_size,12},
    {old_heap_size,0},
    {heap_size,44},
    {bin_vheap_size,0},
    {bin_vheap_block_size,46422},
    {bin_old_vheap_size,0},
    {bin_old_vheap_block_size,46422}] (Timestamp: {1448,829619,911345})

```

如果我们现在分裂出频率服务器，设定最小堆的尺寸为 1024 字（一个更小一些的值也应当足够），此时我们拥有了足够的内存可分配频率而不会触发垃圾回收器：

```

1> dbg:tracer().
{ok,<0.35.0>}
2> {ok, Pid} = gen_server:start_link({local, frequency}, frequency, [],
                                     [{spawn_opt, [{min_heap_size, 1024}]}]).
{ok,<0.38.0>}
3> dbg:p(Pid, [garbage_collection, timestamp]).
{ok, [{matched, nonode@nohost, 1}]}
4> frequency:allocate(), frequency:allocate(), frequency:allocate(),
    frequency:allocate(), frequency:allocate().
{ok,14}

```

进程堆

通过增加短期进程（short-lived process）的 {min_heap_size, Size} 为合适的值，可使其避免执行过程中触发垃圾回收器，并且避免因为需要分配内存而导致堆尺寸增加。如果进程一启动就进行一些内存和 CPU 敏感的活动，然后就终止，那么这样的调整是理想的。当其终止时，全部内存只需一个操作就能释放掉。使用此选项需要当心，因为如果

设置了过大的值将会增加内存消耗并且会减慢你的程序。

Size 是以字为单位的，其具体尺寸取决于处理器的体系结构。在 32 位体系结构中，一个字等于 4 字节（32 位），而在 64 位体系结构中，则是 8 字节（64 位）。使用 `erl` 命令启动 Erlang 运行时系统时，你可以使用 `+hms` 标志来设置所有进程的最小堆尺寸。何时使用 `+hms`？我们的建议是，当你只有相对较少的进程运行于系统中，并且基准测试表明这样设置能够提高性能时。作为一条经验法则，最好还是对各个进程有针对性地设置最小堆尺寸，并且只有当基准测试表明确实有益处时再去做。因为堆尺寸的增加是基于斐波那契数列的，因此最小堆的尺寸将会是序列中下一个大于等于 Size 的数。

虚拟二进制堆

在分裂时的配置选项中，其中有一个选项与垃圾回收有关，而且有助于调优性能，它就是 `{min_bin_vheap_size, VSize}`，作用是配置虚拟二进制堆 VBH（Virtual Binary Heap）的尺寸下限。进程刚开始拥有的 VBH 空间就这么多，垃圾回收过程能够释放掉其中不再使用的二进制数据。只有尺寸大于 64 字节的二进制数据才会被存储到这个所谓的 VBH 中。访问存储于其中的二进制数据需要通过二进制引用，通过这种方式，所有的进程都能读取到同一个二进制数据。你可以设置所有进程的 VBH 尺寸，方法是，用 `erl` 命令启动系统时使用 `+hmb` 标志，但是就像对普通堆那样，你最好三思，尽可能只更改部分进程的 VBH 尺寸，而不是全部。

堆的全量回收

分裂时通过设定 `{fullsweep_after, Number}` 选项，可以指定分代回收要进行多少次后才执行一次全量回收。设定 `Number` 为 0 则禁用分代式垃圾回收机制，这样一来每次都会把年轻堆和老龄堆中的无用数据全部释放。这对于一些内存很少必须严格管理的环境来说有一定的帮助。设为 0 对于另外一种情况也有帮助，那就是如果老龄堆中存在许多大尺寸的二进制数据而你又想频繁地清除它们。设为一个较小的数字也可能是合适的，比如你的数据属于短期型（short-lived），而且基准测试也表明它们正在填满你的堆。Erlang 官方文档建议设置为 10 到 20 之间的值，具体如何选择要靠你根据系统的具体情况进行判断。实际上默认值比这些大得多！

每次休眠你的进程时也会触发全量式垃圾回收。这对于那些偶尔执行内存敏感型计算的进程来说，有助于减少内存消耗。你可以使用 `erlang:system_flag/2` 调用来全局性地设置全量回收值，但我们建议你不要这么做。你可以使用 `process_info/2` BIF 来获取你改变的设置信息：

```
5> process_info(Pid, garbage_collection).
{garbage_collection, [{min_bin_vheap_size, 46422},
```

```
{min_heap_size,1598},
{fullsweep_after,65535},
{minor_gcs,0}}
```

注意, `fullsweep_after` 的默认设置是一个比你预想要高得多的值。我们已经设置了 `min_heap_size` 为 1024, 但是在 shell 命令 5 中, 看起来却是 1598。我们请求的确实是 1024 字, 但 1598 是 VM 堆尺寸所使用的斐波那契递推数列中最接近 1024 的较大数字, 因此最终选择了它而不是 1024。



如果你刚开始尝试各种堆尺寸和垃圾回收设置, 请记住, 内存只有在垃圾回收器被触发后才会被释放。在一些情况下, 进程自己的堆中包含了一些对共享堆中的大二进制数据的引用。对二进制数据的引用相对来说每一个都很小, 因此即使当进程不再引用这些二进制数据了, 如果不触发垃圾回收, 则可能依然会消耗许多内存, 因为在进程自己的堆上还占有着许多空间。这就是为什么每个进程会有自己的虚拟二进制堆的原因, 即计算在共享堆中二进制数据使用的内存总数, 以帮助确保它们更及时地得到释放。在这些情况下, 休眠进程或者使用 `erlang:garbage_collect()` BIF 可能会提供更有效的手段。

另外一个潜在的风险是内存耗尽。举一个例子, 设置一个很大的 `min_heap_size` 并且使用高到危险的 `fullsweep_after` 值 65 536 可能会导致老龄堆由于垃圾回收迟迟不进行和增长, 最终系统甚至还来不及触发一次全量回收就耗尽了内存。要经常对你的系统做压力测试 (stress test), 即使不能跨周进行也要跨天进行饱和测试 (soak test)。

分裂时应该避免使用的可选项

下面的这些选项应当避免使用, 因为它们要么对行为模式无效, 要么被认为是糟糕的编程实践。比如, 使用 `spawn_opt/3` BIF 时可以把 `monitor` 作为一个选项传入, 但在通用型 server 中是不允许这样做的, 结果会导致进程因为参数无效 (badarg) 而终止。另外, 虽然你可以通过使用 `link` 作为选项实现链接, 但是启动行为模式时更好的做法还是通过 `start_link` 来进行。

永远不要使用 `{priority, Level}` (其中的 Level 可以填写原子 low、normal 或者 high) 作为选项去设置进程优先级。改变进程优先级是一件甚至比干涉内存和垃圾回收更危险的事, 因为这会搅乱 VM 的平衡, 并对整个系统的软实时 (soft real-time) 属性带来一系列严重的影响。改变优先级可能导致 VM 的调度器变得奇怪和不公平; 而且已经知道当更高优先级的进程与更低优先级的进程二者的比率满足某些限制时会出现饥饿。更进一步, 当系统负载很高时, 低优先级进程会由于消费消息的速度低于消息产生的速度导致运行时系统耗尽内存。显然, 当你测试你的系统时, 你不会注意到这些问题; 它们会在你的系统运行时处于重负载时突然冒出来咬你一口。还是让运行时系统替你做主吧,

特别是当处理几十万个进程时。别说我没有警告过你!

超时

如果你想限制行为模式有多少时间可以花费在 `init` 函数上, 那么请把 `{timeout, Timeout}` 选项带上。如果超过 `Timeout` 毫秒之后 `init` 回调函数还在执行, 则该进程将被终止, 并且启动函数返回 `{error, timeout}`。这个选项在一些非常特定的情形下比较有用, 通常是某系统中某些进程拥有一些动态的子进程, 而这些子进程负责的又是一些易失易变的资源。不推荐在启动系统时使用这个功能, 我们建议你试着把 `init` 函数中要做的工作尽可能最小化, 这样就不会减慢启动过程。

总结

存在许多选项可以用来控制和监视你的行为模式, 从内置的追踪和日志记录功能开始。你可以随时动态添加通用式追踪和调试触发器或者使用 `fun` (函数) 和 `sys` 模块改变进程的状态, 这些都可以在运行中完成, 不需要重新编译你的代码。这是一个重要的功能, 因为你可以无须重启的情况下把它用在那些你从未见过又处于运行数年之久的系统上。你可以通过阅读 `sys` 模块参考手册页了解更多这方面的信息。

通过在选项中使用内存标志来优化进程性能是一件难事, 因为这要求你对系统进行基准测试, 并且以你从测试过程中了解到的信息作为基础进行优化。极少会有需要你修改默认垃圾回收器设置或者修改堆尺寸的情况。但是如果遇到性能问题时, 你会很感激你读了本章。如果你需要更多信息, 查看 `erlang` 模块手册页中关于 `spawn_opt` BIF 的文档。

接下来是什么

运行中追踪方面的内容暂时讲到这里, 之后在第 10 章实现我们自己的行为模式 (学会相关的幕后工作原理) 时会再继续; 另外性能调优方面的内容也告一段落, 等到第 13 章再继续。在后面的章节中, 我们将聚焦于其余的行为模式, 首先从 FSM 开始, 然后是事件管理器 (event manager)、监督者 (supervisor) 和应用 (application)。记住, 这些行为模式都基于相同的基础, 因此 `sys` 模块以及所有我们已经讨论过的分裂和调试选项对它们都是适用的。

115

有限状态机^{注1}

至此，我们已经掌握如何编写 `gen_server` 了，是时候继续掌握下一个行为模式了。回想当年，Erlang 语言的发明人 Joe Armstrong、Mike Williams 和 Robert Virding 其实是为了实现一个能让他们三个人互打电话问好的软交换（soft telephony switch）原型系统而最终发明了 Erlang。^{注2} 在这个系统中，每一部与交换设备相连的电话都被原型化为一个以 FSM（有限状态机）方式工作的进程。在任何时刻，电话所处的状态（挂断、占线、拨号中、响铃中等）都是由函数表示的，并且能够接收与此状态相对应的事件（来电、拨号、占线、挂断）。

这次原型探索取得的成果之一就是使得 Erlang 成为一门针对构建复杂而且可伸缩的有限状态机（FSM）系统优化过的语言，而这一点恰恰是许多复杂系统中关键的构成部分。开发人员使用有限状态机来编写协议栈（protocol stack）、连接器（connector）、代理（proxy）、工作流系统（workflow system）、游戏引擎（gaming engine）以及模拟器（simulation）等许多东西。所以一点也不例外的，OTP 行为模式中包含了 `gen_fsm`（通用有限状态机）行为模式。

在本章，我们介绍通过纯 Erlang 来实现有限状态机（FSM）。我们把例子的代码分为通用（generic）和专用（specific）两部分，然后迁移到 `gen_fsm` 行为模式中。好消息是所有适用于 `gen_server` 的与并发、错误处理有关的特殊状况处理，同样也适用于 FSM。因此，虽然我们也会提到这些内容，但并不需要像之前一样谈得那么深。毕竟，FSM 的实现本质上是 `gen_server` 的一种变体。

注1 读者请注意，自 Erlang/OTP 20.0 开始，本章介绍的 `gen_fsm` 被设计更好的 `gen_statem` 替代了。但本章介绍核心理念依然适用，因此请不要错过本章！——译者注

注2 影迷们会在 *Erlang the Movie* 看到关于这一交换机的大量片段。影片拍摄时，Erlang 语言仍在演变发展中，所以观众会注意到一些例子中使用的是旧的语法。如果你还没看过这部影片，请前往 YouTube 搜索。这是必看的！

Erlang 风格的有限状态机

在开始深入探讨例子之前，我们先了解一点自动机理论。所谓的 FSM（有限状态机）是一种抽象模型，它由有限数量的状态（state）以及事件（event）构成。当程序处于各个状态时，它可以从环境中接收一些特定的事件——并且只接收这些事件。当事件抵达，而 FSM 处于特定的状态时，程序就会执行一些与当前状态对应的预先决定好的动作，使得当前状态转移为另一个新的状态。然后 FSM 以此新状态等待下一个新的事件。

比如，图 6-1 中所示的 FSM，其中的 *day*（白天）状态可以处理 *eclipse*（日食）和 *sunset*（日落）事件。*eclipse* 事件后 FSM 的状态保持不变，而 *sunset* 事件导致状态转移为 *night*（夜晚）。在 *night* 状态，*sunrise*（日出）事件又使得状态转移回 *day*。对于任何违反顺序出现的事件（例如，在 *day* 状态下出现 *sunrise* 事件），只有当状态转移到能处理的情况后才会处理。

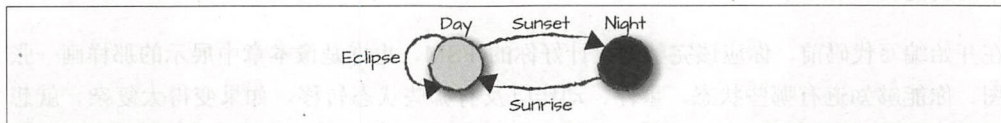


图 6-1: Erlang 中的 FSM

在 Erlang 中，每个状态（state）以一个内含尾递归的函数表示，而事件（event）则以消息（message）表示。因此对于图 6-1 来说，与 *day* 状态对应的代码应该看起来像下面这样：

```

day() ->
    receive
        eclipse -> day();
        sunset  -> night()
    end.
  
```

当接收到一个事件后，FSM 在转移到下一个状态前会执行一个或者多个动作。状态转移是通过调用下一个函数实现的，究竟调用哪一个函数，取决于当前状态与进入的事件的组合。在下面的例子里，事件 *sunrise* 与状态 *night* 的组合将导致函数 `make_roosters_crow/0` 中定义的动作被执行，然后转移到状态 *day*。注意，我们是不允许日食（*eclipse*）发生在夜间（*night*）的。如果 FSM 收到 *eclipse* 事件，该事件将会留在进程的邮箱中，直到 FSM 转移到某个能处理该事件的状态：

```

night() ->
    receive
        sunrise ->
            make_roosters_crow(),
  
```

```
    day()
end.
```

当启动一个 FSM 时，你需要给它一个启动状态，并初始化它。正如下一段示例代码中那样，我们可以通过分裂时传入 `init/0` 函数来完成初始化，其中创建了地球^{注1}（调用了 `create_earth/0`），然后转移到了状态 `day`：

```
start() ->
    spawn(?MODULE, init, []).
```

```
init() ->
    create_earth(),
    day().
```

这就是 Erlang 中编写 FSM 的方式。其中使用的选择性接收（selective receive）、尾递归函数（tail-recursive function）以及能够在分裂进程时初始化等都是使得 FSM 能保持简单的关键。

在开始编写代码前，你应该完整地设计好你的 FSM，也许是像本章中展示的那样画一张图。你能够知道有哪些状态、事件、动作以及有哪些状态转移。如果变得太复杂，就想想能不能把 FSM 分割成几个更小的 FSM，执行时在它们之间来回流动。这些小的 FSM 会更容易实现和维护。



gen_fsm 与 gen_server

小心，新手容易犯这样一种错误——本应使用 `gen_fsm` 却使用了 `gen_server`，不知不觉地把 FSM 状态存储在了循环数据中。当设计系统的时候，问问你自己到底是需要 FSM 还是客户端-服务器的行为模式？如果你是在项目的设计阶段考虑这个问题，那么答案通常很明显。

Coffee FSM

为了能让 Java 迷们开心，我们接下来就用咖啡贩卖机作为 FSM 的例子好了。这是一个嵌入式应用程序，在特定的硬件模块之上建立了接口。我们即将研究其实现，主要涉及三个状态：

- *selection*（选择）状态，允许顾客选择喜欢的咖啡冲泡方式。
- *payment*（付款）状态，允许顾客选择投币付款。
- *remove*（移除）状态，在此状态下，FSM 等待客户从贩卖机中拿走饮料。

120 这些状态由 4 种事件链接起来，这些事件触发相应的动作并使得状态转移。由顾客触发

注 1 对这个函数做基准测试会是一件有趣的事。

的事件包括：

- 针对咖啡做出选择。
- 向槽中投入任意金额的硬币支付刚刚做出的选择。
- 按下取消按钮。
- 从机器中成功拿走咖啡。

请注意,这些事件中的大多数都可以在大多数状态中触发。如果 FSM 处于 *payment* 状态,用户依然可以按下选择咖啡按钮,或者 FSM 正处于 *selection* 状态,用户也总是可以投入硬币。如果事件可以被触发,那么不管当前处于什么状态都必须进行处理。当在某个状态下接收到事件时,在转移到下一个状态前,可以执行一些动作。我们例子中的动作包括：

- 在咖啡贩卖机的 LED 上显示文字。
- 找零给客户。
- 在机器中降下一个空杯子。
- 制作选择的饮料。
- 重启咖啡贩卖机(非用户初始化)。

一个简化版本的 FSM 可以在图 6-2 中看到,其中没有完整描绘所有事件和动作。硬币可以在非 *payment* 状态时投入,取消(cancel)按钮可以在 *selection* 或者 *remove* 状态下按下,这些也可以在启动 FSM 硬件重置时发生。尽管不够完整,但这张图还是提供了一个概览,使我们能看到状态的转移以及触发这些转移的事件。图 6-2 中对各个状态转移过程中会执行的动作做了标注。尖括号中的是动作,事件则以粗体表示。

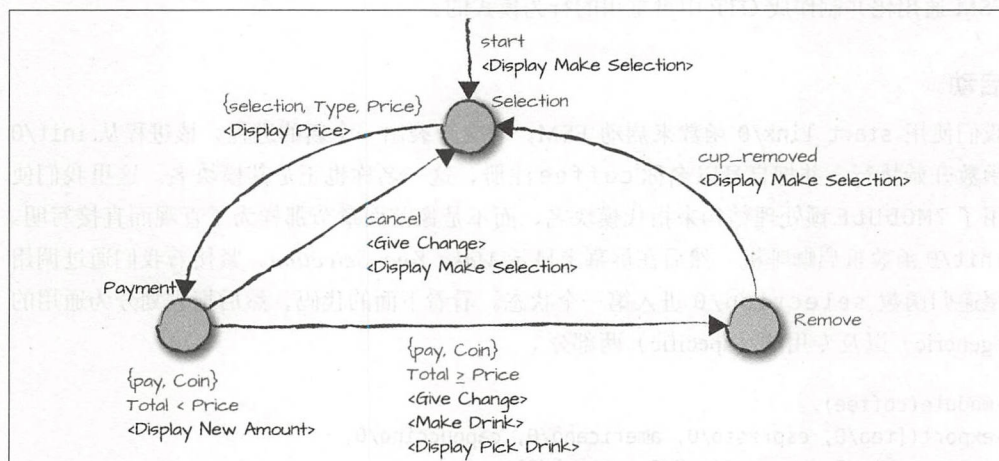


图 6-2: 咖啡机 FSM

121 把这个模型记住之后，让我们开始一步步以纯 Erlang 方式实现这个 FSM 吧。完成之后，我们会把这一实现再改成基于 `gen_fsm behavior` 模块的实现。

硬件桩

像此类需要与传感器及硬件进行交互的嵌入式系统，往往含有 C 语言编写的设备驱动，而我们的 Erlang 代码需要通过一定的接口与其衔接。为使例子简单，我们创建了 `hw.erl` 桩（stub）模块假装实现了这些功能。这个模块既会用在纯 Erlang 版的实现中，也会在使用 `gen_fsm` 行为模式改写的实现中：

```
-module(hw).
-compile(export_all).

display(Str, Arg)    -> io:format("Display:" ++ Str ++ "~n", Arg).
return_change(Payment) -> io:format("Machine:Returned ~w in change~n",[Payment]).
drop_cup()           -> io:format("Machine:Dropped Cup.~n").
prepare(Type)         -> io:format("Machine:Preparing ~p.~n",[Type]).
reboot()              -> io:format("Machine:Rebooted Hardware~n").
```

在 FSM 的实现中你会看到对这一模块的调用。在实际环境下，咖啡机中的传感器会直接调用 `coffee.erl` 模块中的函数，进而导致 `hw.erl` 模块中的这些函数被调用。但出于测试的目的，我们将从 shell 里直接发起调用。明确了这些后，我们接下来进入实现环节。

122 Erlang 版咖啡机

在这一节，我们创建此应用程序中与 Erlang 相关的部分。请记住本例中我们是如何把 FSM 通用化并制作成 OTP 中可重用的行为模式的。

启动

我们使用 `start_link/0` 函数来启动 FSM。它会分裂出一个新的进程，该进程从 `init/0` 函数开始执行，并把自身以名称 `coffee` 注册，这一名称也正是其模块名。这里我们使用了 `?MODULE` 预处理结构来指代模块名，而不是像前面章节那样为了直观而直接写明。`init/0` 函数重启咖啡机，然后在屏幕上显示 *Make Your Selection*。紧接着我们通过调用尾递归函数 `selection/0` 进入第一个状态。看看下面的代码，然后把它划分为通用的（generic）以及专用的（specific）两部分：

```
-module(coffee).
-export([tea/0, espresso/0, americano/0, cappuccino/0,
        pay/1, cup_removed/0, cancel/0]).
-export([start_link/0, init/0]).
```



```

- start_link() ->
    {ok, spawn_link(?MODULE, init, [])}.

init() ->
    register(?MODULE, self()),
    hw:reboot(),
    hw:display("Make Your Selection", []),
    selection().

```

本例里属于通用的代码做了加粗处理，包括分裂进程（使其运行 `init/0`）、注册名称、转移至第一个状态。专用于咖啡机的代码则包括：具体的进程名称是什么、回调模块是哪一个、`init/0` 中那些与硬件紧密相关的调用，以及调用时具体传入的参数。第一个状态是什么也是专用的，可能传递给该状态的一切循环数据也是专用的。在我们的例子里，启动时不需要状态。

各种事件

两组 `client` 函数都会生成事件，这些事件会以异步调用的方式传递给咖啡机 FSM。第一组，即前 4 个函数，告诉 FSM 用户选择了什么饮料，以及价格。当硬件传感器发现杯子被拿走的时候，就会触发 `cup_removed` 事件。如果硬币被投入了，`pay/1` 函数被调用，并把硬币的面值作为参数传入。最后，当“取消”按钮被按下的时候，`cancel/0` 函数被调用。正如我们前面提到的，这些事件可以在任意状态下触发。你无法阻止用户在饮料正在准备时去按“取消”按钮，也无法阻止用户不选择饮料就直接投币。这些 `client` 函数如下所示：

123

```

%% Client Functions for Drink Selections

tea()          -> ?MODULE ! {selection, tea,      100}.
espresso()     -> ?MODULE ! {selection, espresso, 150}.
americano()    -> ?MODULE ! {selection, americano, 100}.
cappuccino()   -> ?MODULE ! {selection, cappuccino, 150}.

%% Client Functions for Actions

cup_removed()  -> ?MODULE ! cup_removed.
pay(Coin)      -> ?MODULE ! {pay, Coin}.
cancel()       -> ?MODULE ! cancel.

```

在这些 `client` 函数里，具体的与事件相关的标签以及数据（例如价格）都是专用的。而发送事件给 FSM 以及可能由此产生的同步或者异步调用则是通用的。在我们的例子里，调用全都是同步的。对于同步调用，返回的值是什么也是专用的，但调用使用的协议以及为了接收回应所使用的 `receive` 语句却是通用的。

selection 状态

在 `init/0` 函数中,当完成了咖啡机初始化后,我们转移到第一个状态,即 *selection* 状态,顾客在此状态下可以选择饮料。一旦收到事件 `{selection, Type, Price}`,我们显示出饮料的价格,并且移动到下一个状态 *payment*。进入此状态的同时,我们传入参数 `Type`(饮料类型)、`Price`(价格),以及 `Paid`(已付金额),这些参数的初始值为 0。这三个参数是 *payment* 状态下循环数据的组成部分。

如果客户插入硬币后没有做出选择,我们必须将钱退回。如果客户按下取消按钮,我们需要从进程邮箱中删除该事件,确保不会在稍后的状态中意外接收到该事件:

```
%% State: drink selection
selection() ->
    receive
        {selection, Type, Price} ->
            hw:display("Please pay:~w",[Price]),
            payment(Type, Price, 0);
        {pay, Coin} ->
            hw:return_change(Coin),
            selection();
        _Other -> % cancel
            selection()
    end.
```

124 状态与事件的组合对应了特定的动作和状态转换。其中,接收事件、处理状态转换和存储循环数据的部分都属于通用类代码。而处理事件,即更新显示、返回硬币,以及决定下一个状态是哪一个等则属于专用类代码。

payment 状态

当客户选择饮料后,可以付款或取消选择。每投入一枚硬币将生成一个 `{pay, Coin}` 事件, `Coin` 是投入的金额,这个数会加到总额中。如果总额大于或等于饮料的价格,则代码将触发动作并转移到 *remove* 状态。如果投入的金额依然不足,则将更新待支付的剩余金额数,并且 FSM 仍保持在 *payment* 状态。如果客户按下取消按钮,则向其返回全部已付款,并且 FSM 返回到 *selection* 状态。至于任何其他事件——更具体地说,即按任何选择按钮——都被忽略。我们忽略事件的方式是重新调用当前状态:

```
%% State: payment

payment(Type, Price, Paid) ->
    receive
        {pay, Coin} ->
            if
```



```

Coin + Paid >= Price ->
    hw:display("Preparing Drink.",[]),
    hw:return_change(Coin + Paid - Price),
    hw:drop_cup(), hw:prepare(Type),
    hw:display("Remove Drink.", []),
    remove();
true ->
    ToPay = Price - (Coin + Paid),
    hw:display("Please pay:~w",[ToPay]),
    payment(Type, Price, Coin + Paid)
end;
cancel ->
    hw:display("Make Your Selection", []),
    hw:return_change(Paid),
    selection();
_Other -> %selection
    payment(Type, Price, Paid)
end.

```

与 *selection* 状态一样，接收事件、状态转换和存储循环数据都属于通用类代码。而专用类代码则包括事件本身、由此引发的操作以及下一个状态。存储循环数据本可以借助一个记录 (record) 式变量完成，但是考虑到不同的状态需要不同数量的参数，此例采用了更为直观的方式。

125

remove 状态

当咖啡钱已付并已冲泡完成时，FSM 进入 *remove* 状态。把它作为一个单独的状态是因为机器必须等待用户取走杯子后才能制作其他饮品。当用户取走杯子时，传感器将触发 *cup_removed* 事件并重置显示内容，可以转换到 *selection* 状态，重新开始。我们无法阻止客户在处于 *remove* 状态时投入硬币，如果出现这种状况，付款必须退回。而客户按取消或选择按钮，同样也属于必须忽略的事件：

```
%% State: remove cup
```

```

remove() ->
    receive
        cup_removed ->
            hw:display("Make Your Selection", []),
            selection();
        {pay, Coin} ->
            hw:return_change(Coin),
            remove();
    end
end

```

```

        _Other -> % cancel/selection
            remove()
    end.

```

在开始阅读下一节关于 FSM 行为模式的内容之前，请下载代码和桩模块尝试一下。一边实验一边思考基于 Erlang 的 FSM 有哪些可能的实现方案。识别其中哪些部分是专用类代码，哪些部分是通用类代码？在通用类代码中，如何将通用代码打包为一个基于回调的库模块？

gen_fsm

为了区分 FSM 中的通用与专用功能，我们将采用与介绍 `gen_server` 时相同的方法。表 6-1 列出了 FSM 中的通用部分和专用部分。

126 表 6-1: FSM 的通用代码与专用代码

通用的	专用的
• 分裂 FSM 进程	• 初始化 FSM 状态
• 存储循环数据	• 循环数据的具体内容
• 向 FSM 发送事件	• 事件的具体内容
• 发送同步请求	• 处理事件 / 请求的具体过程
• 接收回复	• FSM 状态
• 超时	• 状态转变
• 停止 FSM	• 清理

分裂 FSM 进程，确保其正确启动，并且注册——这些是固定的内容，不会随着实现改变而改变。而会改变的则是本地或全局注册时具体的名称、调试选项以及初始化所需的参数。初始化 FSM 的过程属于专用的，包括确定初始状态和绑定循环数据。此二者都会被返回给 `gen_fsm` 的接收 - 求值循环（receive-evaluate loop），数据与状态通常存储于其中。

向 FSM 发送同步或异步的事件和请求以及接收回复等都属于通用的。而事件和请求的内容以及如何根据当前 FSM 状态进行处理等则属于专用的。

所有的状态、对应执行的动作、选择转移到哪个后续状态以及更新循环数据等这些全都是专用的。处理客户端和 FSM 自身的超时则是通用的。然而另一方面，超时触发后将会发生什么则属于专用的。最后，停止 FSM 是通用的，而在停止之前的清理内容则是专用的。

我们可以将 FSM 看作在 `gen_server` 的基础上增加状态处理过程后扩展而得的。消息对应事件，而接收消息的回调函数对应状态。所有的通用代码都放在一个名为 `gen_fsm` 的库模块中，而所有的专用部分都放在一个回调模块中。该架构如图 6-3 所示，可以与图 4-1 进行比较。

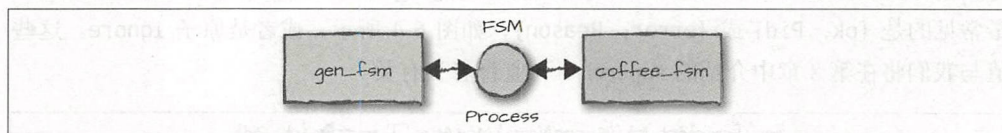


图 6-3: FSM回调模块

一个基于行为模式的例子

127

围绕咖啡机的例子，我们来看看 `gen_fsm` 行为模式模块的各个 API 及回调函数。我们将尝试启动和停止 `gen_fsm`，并探索各个同步 / 异步事件。当我们通读代码时，还会将 `gen_fsm` 行为模式与 `gen_server` 行为模式进行比较。如果你想参与练习，请从本书对应的项目仓库中下载代码。

启动 FSM

每个行为模式回调模块都是以 `module`、`behavior` 和 `export` 指令开始的。其中还包含了所有状态回调函数。除此之外，尽管不是强制性的，但实践经验表明，把所有生成事件的客户端函数都集中放在其中是一个较好的做法。我们的 `coffee_fsm` 模块如下所示：

```
-module(coffee_fsm).  
-behavior(gen_fsm).  
  
-export([start_link/0, stop/0]).  
-export([init/1, terminate/3, handle_event/3]).           % 回调函数  
-export([selection/2, payment/2, remove/2]).             % 各种状态  
-export([americano/0, cappuccino/0, tea/0, espresso/0, % 客户端函数  
        pay/1, cancel/0, cup_removed/0]).
```

`-behavior` 指令中填写原子 `gen_fsm`，这样做的好处是如果漏了实现或导出回调函数，编译时会收到警告。导出的函数包括 `start` 和 `stop` 函数及其对应的回调、客户端函数以及状态回调函数等。

调用 `gen_fsm:start_link/4` 就能启动咖啡机，这一过程分裂出 FSM 进程并将其与父级进程相链接。调用后返回元组 `{ok, Pid}`，其中 `Pid` 标识的是成功分裂出来的进程，否则如果出现问题，则返回 `{error, Reason}`。后文中我们会介绍具体可能出现哪些问题，

现在，我们把目光聚焦在当前的例子上。

与所有 OTP 行为模式一样，我们更喜欢把对 `start_link/4` 的调用包装为回调模块中的客户端函数。在我们的示例中，将其命名为 `coffee_fsm:start_link/0`，但它其实可以是任何你喜欢的名字。真正重要的是它最终会调用 `gen_fsm:start_link` 并返回其返回值：最常见的是 `{ok, Pid}` 或 `{error, Reason}`，如图 6-4 所示，或者是原子 `ignore`。这些值与我们将在第 8 章中介绍的 supervisor（监督者）有关。

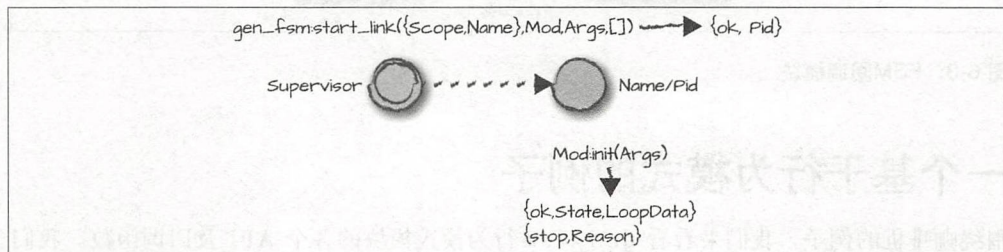


图 6-4: 启动一个 `gen_fsm`

128 `gen_fsm` 进程一分裂就会立刻调用回调模块中的 `init/1` 函数。与 `gen_server` 一样，所有专用的初始化代码都在这个函数中。在我们的示例中，包括重新启动硬件、重置显示屏，并返回一个格式为 `{ok, StartState, LoopData}` 的元组，其中 `StartState` 表示 FSM 收到第一个事件时所处的状态。`LoopData` 包含传递给状态回调函数的数据。在这个例子中，我们还捕获了 `exit` 信号，这样做的原因等到本章后面我们介绍“终止”部分时再进行说明：

```
start_link() ->
    gen_fsm:start_link({local, ?MODULE}, ?MODULE, [], []).

init([]) ->
    hw:reboot(),
    hw:display("Make Your Selection", []),
    process_flag(trap_exit, true),
    {ok, selection, []}.
```

在我们的例子中，`StartState` 是 `selection`，并且没有使用 `LoopData`，所以我们简单地返回空列表值 `[]`。当 `init/1` 回调将控制权返回给通用模块时，`gen_fsm:start_link` 同步调用随即返回。

我们在本地注册该进程，并在设置回调模块处使用了 `?MODULE` 宏，该宏将在编译时被原子 `coffee_fsm` 替换。我们将 `[]` 作为参数传递给 `init/1` 回调函数，没有设置任何可选项。

下面的函数与 `gen_server` 模块导出的那些函数的功能相仿，用于启动 FSM 进程：

```
gen_fsm:start_link(NameScope,Mod,Args,Opts)
gen_fsm:start(NameScope,Mod,Args,Opts)
gen_fsm:start_link(Mod, Args, Opts)
gen_fsm:start(Mod, Args, Opts) -> {ok, Pid}
                                {error, Reason}
                                ignore

Mod:init/1 -> {ok, NextState, LoopData}
            {stop, Reason}
            ignore
```

`NameScope` 影响我们采用何种方式注册行为模式。与 `gen_server` 一样，可以将其设置为 `{local, Name}`、`{global, Name}` 或 `{via, Module, ViaName}`，其中 `via` 开头的元组指向的是用户自定义的进程注册表模块，其导出与 `global` 模块相同的 API，这方面的内容在之前第 4 章的“全局化”一节中已介绍过。使用 `start` 函数可避免使 FSM 进程链接到其父进程，还可以决定不做进程注册。`Opts`（第 5 章所述）同样可以被传入，其中包括超时、调试、分裂等相关选项。不过在此，我们只是传入了一个空列表作为 `Opts`。

◀ 129

如果 `init/1` 回调执行时出现问题，可以直接终止也可以返回元组 `{stop, Reason}`。这样一来错误将传播给调用 `gen_fsm` 启动函数的父进程（通常这个启动函数是写在回调模块中的，提供外部调用），导致它也终止。如果父进程恰好是一个 `supervisor`，它将依次终止所有子进程并中止启动过程。虽然错误随时可能出现，包括在系统运行过程中，但对于 `init/1` 回调函数中出错这种情形，默认情况而言系统无力恢复。

你在 shell 中测试 FSM 时最常遇到的错误提示是 `{error, {already_started, Pid}}`。当具有相同注册名称的其他进程已存在时会发生此错误：

```
1> coffee_fsm:start_link().
Machine:Rebooted Hardware
Display:Make Your Selection
{ok,<0.38.0>}
2> coffee_fsm:start_link().
{error,{already_started,<0.38.0>}}
```

如果想让 `supervisor` 在碰到 `init/1` 失败时继续启动其他 worker，则你应返回原子 `ignore`。这样做，`supervisor` 不会中止启动过程，而是会保存子进程规格（`child specification`）并继续启动其他行为模式。我们将在第 8 章介绍 `supervisor` 时更详细地介绍 `ignore` 和 `stop` 选项。

现在我们继续看看下面的例子，从中你应该能理解不同做法产生的不同效果。请注意导

致 `start` 和 `start_link` 函数调用终止的原因。我们在这个例子中省略了模块头部内容。如果要查看它们，请从本书的代码库中下载 `test_fsm.erl` 模块：

```
start_link(TimerMs, Options) ->
    gen_fsm:start_link(?MODULE, TimerMs, Options).
start(TimerMs, Options) ->
    gen_fsm:start(?MODULE, TimerMs, Options).

init(0) ->
    {stop, stopped};
init(1) ->
    {next_state, selection, []};
init(TimerMs) ->
    timer:sleep(TimerMs),
    ignore.
```

让我们来运行代码吧。在第一组测试中，我们通过返回 `{stop, Reason}` 来停止 FSM：

```
1> test_fsm:start_link(0, []).
** exception exit: stopped
2> test_fsm:start(0, []).
{error,stopped}
```

注意当 `shell` 与行为模式链接及不链接时的表现差异。

在 `shell` 命令 3 和 4 中，我们使用 `test_fsm:init(1)` 调用初始化 FSM，这意外地致使回调模块返回的元组中包含的第一个元素是 `next_state` 而不是 `ok`。对于 FSM 后端模块而言这属于无效的返回值，而此类错误作者自己犯过很多次：

```
3> test_fsm:start_link(1, []).
** exception exit: {bad_return_value,{next_state,selection,{}}}
4> test_fsm:start(1, []).
{error,{bad_return_value,{next_state,selection,{}}}}
```

每当你返回了不符合预定协议的控制元组时，行为模式模块将以 `bad_return_value` 原因终止。

当阅读本例时，请确保你了解了对于 `shell` 进程与 FSM 链接以及不链接两种状况下 `EXIT` 信号传播过程的影响差异。在 `shell` 命令 5 中，我们将一个 1000 毫秒的参数传递给 `init/1`，使其在这段时间内睡眠，并且同时设定了 `timeout` 可选项为 100 毫秒；这触发了启动过程的超时，导致产生 `{error, timeout}` 元组。无论进程是否链接到 `shell` 进程，都将返回这一个值：

```
5> test_fsm:start_link(1000, [{timeout, 100}]).
{error,timeout}
```


在我们的最后一组测试中，即 shell 命令 6 和 7 中，init/1 函数返回 ignore。这不会导致行为模式异常终止，因而也不会产生 EXIT 信号的传播：

```
6> test_fsm:start_link(2, []).
```

```
ignore
```

```
7> test_fsm:start(2, []).
```

```
ignore
```

尽管这些示例中使用的都是 gen_fsm 行为模式，但对所有 OTP 工人进程（worker）都是有效的。

关于 FSM 的启动和初始化我们已介绍得足够多了。接下来让我们探讨生活中重要的事——如何煮咖啡。

发送事件

启动咖啡 FSM 后，我们需要能够定义状态并发送同步和异步事件。处理这些事件将触发状态的转换。事件通常是通过回调模块中定义的客户端函数发送的。让我们从研究 FSM 中的异步事件开始，了解它们在不同状态下的处理方式。

异步事件

异步事件的发送是使用 gen_fsm:send_event(Name, Event) 库函数完成的。它会将 Event 发送给 FSM 进程，而 FSM 进程将会在回调模块中的回调函数 State(Event, LoopData) 中处理它。处理请求后，State/2 函数将返回新的循环数据以及下一个状态 (next_state) 或者停止 (stop)（参见图 6-5）。

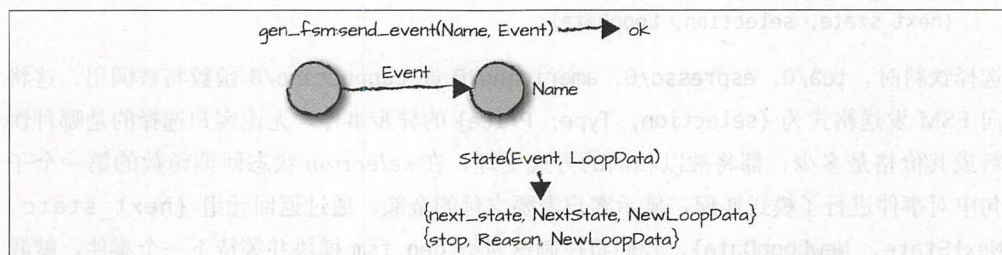


图 6-5：发送事件

我们的 FSM 中的事件函数可以分为两类。第一类是客户选择饮料。选择时会发送格式为 {selection, Type, Price} 的事件，其中 Type 是原子 tea、espresso、americano^{注1}

注1 美式咖啡（Americano coffee）是在浓缩咖啡中加入大量水制作而成的，这是我们的最爱，所以不能去掉。

或 cappuccino。Price 则是 100 或 150：

```
tea()      -> gen_fsm:send_event(?MODULE,{selection,tea,100}).
espresso() -> gen_fsm:send_event(?MODULE,{selection,espresso,100}).
americano() -> gen_fsm:send_event(?MODULE,{selection,americano,150}).
cappuccino()-> gen_fsm:send_event(?MODULE,{selection,cappuccino,150}).
```

第二类事件则包括一些用户动作，例如插入硬币、按下取消按钮或拿走杯子等。事件并非只能由静态值构成。注意，在 `pay/1` 函数中，我们把传入的变量作为事件的一部分——投入的硬币金额被绑定到 `Coin` 上并传入事件 `{pay, Coin}`：

```
132 pay(Coin)      -> gen_fsm:send_event(?MODULE,{pay, Coin}).
    cancel()      -> gen_fsm:send_event(?MODULE, cancel).
    cup_removed() -> gen_fsm:send_event(?MODULE, cup_removed).
```

定义状态

FSM 中的状态是以回调函数的形式定义的，函数名即状态名，`Event` 作为第一个参数，`LoopData` 则是第二个参数。记住，状态回调函数是在回调模块中定义的，因而必须导出。我们看到的第一个状态是 `selection`，在这一状态下客户被提示选择一种饮料。当我们启动 FSM 时，`init/1` 函数返回的初始状态正是 `selection`：

```
selection({selection,Type,Price}, _LoopData) ->
    hw:display("Please pay:~w",[Price]),
    {next_state, payment, {Type, Price, 0}};
selection({pay, Coin}, LoopData) ->
    hw:return_change(Coin),
    {next_state, selection, LoopData};
selection(_Other, LoopData) ->
    {next_state, selection, LoopData}.
```

选择饮料时，`tea/0`、`espresso/0`、`americano/0` 或 `cappuccino/0` 函数将被调用。这将向 FSM 发送格式为 `{selection, Type, Price}` 的异步事件。无论客户选择的是哪种饮料或其价格是多少，都将被以相同的方式受理。在 `selection` 状态回调函数的第一个子句中对事件进行了模式匹配，显示客户需要支付的金额。通过返回元组 `{next_state, NextState, NewLoopData}`，我们将控制返回给 `gen_fsm` 模块并等待下一个事件。就我们的例子而言，`NextState` 绑定到了 `payment` 状态，并将 `LoopData` 绑定到一个代表了用户所选咖啡类别 (`Type`)、对应价格 (`Price`) 以及已付金额 (初始值为 0) 的元组。请留意传入的循环数据——它是在 `init/1` 回调函数中设定的空列表，因而我们忽略了它，但是我们为下一个状态创建了新的循环数据。

如果客户在 `selection` 状态向咖啡机投入硬币会发生什么？在示例中，我们调用

hw:return_change/1 使得 FSM 将硬币退回给客户，并继续保持在 *selection* 状态，不改变循环数据（其实就是空列表）。如果你更倾向于留下硬币，那可以删除这行代码。或者，如果你打算实现豪华版的咖啡机，可以添加功能来阻止用户投币。

当处于 *selection* 状态时，客户可以生成不需要任何操作或状态更改的事件。例如按下取消按钮或触发拿走杯子的传感器等，这些事件是需要处理的，但是就此种情形而言既不需要改变当前状态也不需要改变循环数据，因此处理过程仅仅是忽略它们。当客户按下取消按钮时将触发一个 `selection(cancel, [])` 调用，而我们并没有写下相应的代码，所以将导致运行时错误，原因是没有匹配的函数子句。

133

如果客户选择美式咖啡，FSM 将显示待付金额并转入 *payment* 状态，期待着下一个事件：

```
payment({pay, Coin}, {Type, Price, Paid}) when Coin+Paid < Price ->
    NewPaid = Coin + Paid,
    hw:display("Please pay:~w",[Price - NewPaid]),
    {next_state, payment, {Type, Price, NewPaid}};
payment({pay, Coin}, {Type, Price, Paid}) when Coin+Paid >= Price ->
    NewPaid = Coin + Paid,
    hw:display("Preparing Drink.",[]),
    hw:return_change(NewPaid - Price),
    hw:drop_cup(), hw:prepare(Type),
    hw:display("Remove Drink.", []),
    {next_state, remove, null};
payment(cancel, {_Type, _Price, Paid}) ->
    hw:display("Make Your Selection", []),
    hw:return_change(Paid),
    {next_state, selection, null};
payment(_Other, LoopData) ->
    {next_state, payment, LoopData}.
```

客户现在需要支付咖啡费用。每次投入硬币时，都会生成 `{pay, Coin}` 事件。我们将硬币的金额加到付款总额上，如果总额小于饮料的价格，则显示剩余的待付金额。通过将 *payment* 作为下一个状态返回，我们将 FSM 保持在该状态，并更改循环数据以反映迄今已支付的金额。

如果客户已经投入了足够的钱，我们会触发一连串的行动，从改变屏幕显示开始，表明我们正在准备饮料。找零并放下杯子，并开始煮咖啡，并且只有等到饮料做好时才从同步调用 `hw:prepare(Type)` 中返回。然后，提示客户拿走饮料并将控制返回到 `gen_fsm` 控制循环，进入 *remove* 状态。

客户为咖啡付款时，可能改变主意，按下取消按钮。如果他们这样做，我们会把屏幕显示改为“请选择（Make Your Selection）”，退还他们已经投入的所有硬币，并维持下一

个状态依然是 *selection*。如果客户触发了拿走杯子的传感器或按下选择任何一种饮料的按钮，我们都忽略该事件并保持在 *payment* 状态。

现在让我们假设客户已经为一杯饮料付了钱，还收到了找零，并且饮料也煮好了。那么 FSM 将处于 *remove* 状态：

```
remove(cup_removed, LoopData) ->
    hw:display("Make Your Selection", []),
    {next_state, selection, LoopData};
remove({pay, Coin}, LoopData) ->
    hw:return_change(Coin),
    {next_state, remove, LoopData};
remove(_Other, LoopData) ->
    {next_state, remove, LoopData}.
```

当客户取走杯子时，咖啡机中的传感器将被触发。这导致 `coffee_fsm:cup_removed()` 调用，进而导致一个子句处理 `cup_removed` 事件。咖啡机将显示屏更新为“请选择(Make Your Selection)”，并在函数返回时，将下一个状态设置为 *selection*。在 *remove* 状态下，客户还可以投入硬币，这会触发第二个子句中退回硬币的动作，客户还可以按下取消或者饮料选择按钮，触发第三个子句中的忽略。

揭晓真相的时刻到了。我们能喝到想喝的咖啡吗？来测试我们的程序看看它是否有效。当编译你的行为模式时，正如我们在第 4 章的“`gen_server`”一节中所看到的那样，在编译本章中的代码时，会收到缺少 `code_change/3` 回调的警告。我们将在第 12 章介绍这一点。

为了更好地了解发生了什么，我们将使用第 5 章“追踪与记录”一节中介绍的 OTP 内置的调试选项。启动 FSM，选择茶 (tea)，然后改选美式咖啡 (Americano coffee)，并投入两个金额为 100 的硬币。我们获得找零，并在等待拿走杯子的时候，又投入一个 50 的硬币，意在测试 FSM。在你一步步实验的过程中，可以通过 `*DBG*` 前缀来区分哪些是你收到提示后输入的（例如 1>），哪些是调试器打印输出的。`hw.erl` 模块中使用 `io:format/2` 输出的内容由一段提示开头——指明了其代表系统的哪一部分 (Display: 或 Machine:)，而剩下的部分则来自函数调用返回的结果值：

```
1> {ok, Pid} = coffee_fsm:start_link().
Display:Make Your Selection
{ok,<0.68.0>}
2> sys:trace(Pid, true).
ok
3> coffee_fsm:cancel().
*DBG* coffee_fsm got event cancel in state selection
```



```

ok
*DBG* coffee_fsm switched to state selection
4> coffee_fsm:tea().
*DBG* coffee_fsm got event {selection,tea,100} in state selection
ok
Display:Please pay:100
*DBG* coffee_fsm switched to state payment
5> coffee_fsm:cancel().
*DBG* coffee_fsm got event cancel in state payment
ok
Display:Make Your Selection
Machine:Returned 0 in change
*DBG* coffee_fsm switched to state selection
6> coffee_fsm:americano().
*DBG* coffee_fsm got event {selection,americano,150} in state selection
ok
Display:Please pay:150
*DBG* coffee_fsm switched to state payment
7> coffee_fsm:pay(100).
*DBG* coffee_fsm got event {pay,100} in state payment
ok
Display:Please pay:50
*DBG* coffee_fsm switched to state payment
8> coffee_fsm:pay(100).
*DBG* coffee_fsm got event {pay,100} in state payment
ok
Display:Preparing Drink.
Machine:Returned 50 in change
Machine:Dropped Cup.
Machine:Preparing americano.
Display:Remove Drink.
*DBG* coffee_fsm switched to state remove
9> coffee_fsm:pay(50).
*DBG* coffee_fsm got event {pay,50} in state remove
ok
Machine:Returned 50 in change
*DBG* coffee_fsm switched to state remove
10> coffee_fsm:cup_removed().
*DBG* coffee_fsm got event cup_removed in state remove
ok
Display:Make Your Selection
*DBG* coffee_fsm switched to state selection
11> sys:trace(Pid, false).
ok

```

看起来运转正常，可以松一口气了！

超时

我不知道你有没有经历过这种情形，想象一下你正耐心地排队等待买咖啡。排队的时候，你脑子里已决定好要选哪一种咖啡，并准备好了零钱。但你前面那些人却慢悠悠的。他们要把所有选项都看一遍后，才慢悠悠地做出选择。而且要等到屏幕显示出价格，他们才会从钱包或口袋里找零钱，而且不是足够的零钱，而真的只是“零钱”！他们投入一分钱，然后再伸回口袋里寻找更多的一分钱，直到他们找不到。然后，他们开始寻找五分钱和一角钱。这种状况可能会愈演愈烈，不仅仅是作者们这么认为。幸运的是，我们现在控制着咖啡机，因而我们有机会阻止这种不当的行为。

在 FSM 内指定超时值时既可以指定为整数（以毫秒为单位），也可以指定为原子 `infinity`。可以将超时值包含在 `init/1` 和 `State` 回调函数中。当触发超时，事件将被发送到 FSM 当前处于的状态。由于我们能够控制咖啡机的代码，所以可以设定一旦客户在两次投入硬币之间超过 10 秒就触发超时，这可以给那些慢悠悠的人带来一些压力。首先，我们通过重构 `payment` 状态来引入超时：

```
-define(TIMEOUT, 10000).
...

selection({selection,Type,Price}, _LoopData) ->
...
{next_state, payment, {Type, Price, 0}, ?TIMEOUT};

payment({pay, Coin}, {Type,Price,Paid}) when Coin+Paid >= Price ->
...
{next_state, remove, []};
payment({pay, Coin}, {Type,Price,Paid})
when Coin+Paid < Price ->
...
{next_state, payment, {Type, Price, NewPaid}, ?TIMEOUT};
payment(timeout, {Type, Price, Paid}) ->
hw:display("Make Your Selection", []),
hw:return_change(Paid),
{next_state, selection, []};
payment(_Other, LoopData) ->
{next_state, payment, LoopData, ?TIMEOUT}.
```

投硬币的客户现在必须快一点。如果投入一枚硬币花费超过 10 秒，他们的选择就将被取消，并且钱被返还。一个已知的风险是，他们也许会发现只要按下任意一个饮料选择按钮，就会得到额外的 10 秒，但我们假设，他们会专注于寻找他们的下一分钱，因而无须担心这一问题。

如果想减少 `gen_fsm` 占用的内存空间，可以通过在超时值处设为原子 `hibernate` 来实现这一目的。注意只有在你不希望 FSM 在一段时间内接收到事件时，并且基准测试表明你碰到了内存方面的问题时，才考虑使用 `hibernate`。除此之外，我们还可以停止 FSM，具体内容会在本章稍后介绍：

```
gen_fsm:send_event(NameScope ,Event) -> ok
```

```
Mod:State/2 -> {next_state, NextState, NewLoopData}
               {next_state ,NextState, NewLoopData, Timeout}
               {next_state, NextState, NewLoopData, hibernate}
               {stop, Reason, NewLoopData}
```

任意状态下的异步事件

如果要发送异步事件，并且不论当前处于什么状态都希望接收到，可以使用 `send_all_state_event/2` 调用。如果要执行诸如格式化、打印循环数据或停止 FSM 之类的操作，这可能很有用。事件会作为第一个参数传递给 `handle_event/3` 回调函数，该函数执行该操作后将 `{next_state, NextState, NewLoopData}` 元组返回给 `gen_fsm` 控制循环（参见图 6-6）。

137

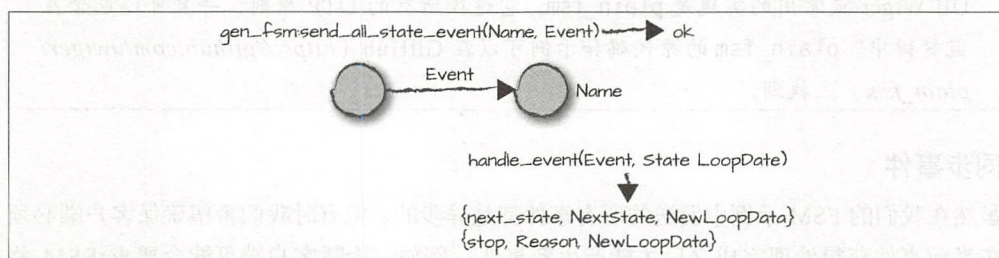


图 6-6：发送事件给全部状态

与 `gen_server` 一样，`handle_info/3` 回调函数处理所有的非 OTP 类消息，如 `EXIT` 信号、监视器消息、使用 `Pid!Msg` 构造发送的消息等。`handle_info/3` 回调返回与 `handle_event/3` 和 `State/2` 相同的控制元组：

```
gen_fsm:send_all_state_event(NameScope ,Event) -> ok
```

```
Mod:handle_info/3,
Mod:handle_event/3 -> {next_state, NextState, NewLoopData}
                      {next_state ,NextState, NewLoopData, Timeout}
                      {next_state, NextState, NewLoopData, hibernate}
                      {stop, Reason, NewLoopData}
```

选择性接收

`gen_fsm` 行为模式模块没有提供选择性接收功能。在跨不可靠的分布式网络运行的复杂的有限状态机系统中，事件只有少数情况会按顺序到达。想象一下，当你在 `night` 状态的时候收到了一个 `sunset` 事件！你可以要么在循环数据中缓冲这些事件，等达到某个知道如何处理它们的状态时处理它们，或者借助一个额外的状态信息，使失序事件恢复有序。但相较于使用选择性接收，这两种解决方案都会导致不必要的复杂性，因为选择性接收会简单地将事件留在进程邮箱中，直到在实际处理它们的状态下匹配为止。

之所以欠缺这一功能是有意为之的，这样消息会按照它们到达的顺序被处理，确保不会出现由于某个消息不被匹配而导致内存泄漏。`gen_fsm` 行为中的事件以先进先出（FIFO）的方式处理，并在读取时从接收进程的邮箱中删除。

如果你想避免由于消息必须按抵达顺序处理而导致的复杂性，有两种解决方法。你可以实现一套自己的选择性接收式 FSM 行为，具体做法我们将在第 10 章中介绍。或者你可以使用其他人已经实现的选择性接收式 FSM 行为。在撰写本文时，Ulf Wiger 最常用的实现是 `plain_fsm`。它遵循所有的 OTP 原则，并且可以包含在监督树中。`plain_fsm` 的源代码和示例可以在 GitHub (https://github.com/uwiger/plain_fsm) 上找到。

同步事件

虽然在我们的 FSM 示例中发送的所有事件都是异步的，但有时我们希望确保客户端必须在当前事件获得处理完成之后才能产生新事件。例如，诊断客户端可能会要求 FSM 将特定值设置到硬件寄存器中，并且在 FSM 指明已设置成功之前不采取任何进一步措施。如图 6-7 所示，这时可以使用 `sync_send_event/2`（或 `sync_send_all_state_event/2`）调用。

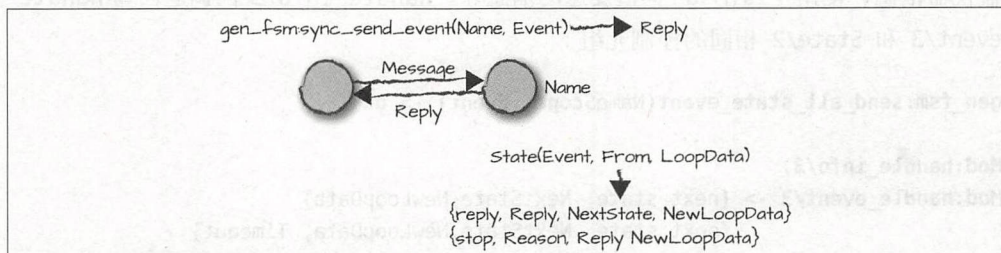


图 6-7：同步事件

此函数及其对应的回调函数所处的地位位于 `gen_server` 中的 `call/2` 与 `handle_call/3`

函数式风格与 FSM 的异步事件与异步事件处理风格之间，仿佛是“中间派”。事件是在 `State(Event, From, LoopData)` 回调中处理的，其中 `From` 是一个元组，包含了客户端和请求的引用。该回调不返回 `next_state` 元组，而是返回一个格式为 `{reply, Reply, NextState, NewLoopData}` 的元组。`Reply` 会被发送回客户端，成为 `gen_fsm:sync_send_event/2` 调用的返回值。

和在 `gen_server` 中一样，我们可以通过调用 `gen_fsm:reply(From, Reply)` 来把 `Reply` 发送回 `From` 标识的原始调用者，注意，需要在 `State/3` 回调函数中返回 `{next_state, NextState, NewLoopData}`。

使用 `gen_fsm:sync_send_all_state_event/2` 函数（参见图 6-8）能向 FSM 发送同步请求，无论 FSM 当前处于何种状态。事件会在 `handle_sync_event/4` 回调函数中被处理，该函数要么通过 `From` 返回 `Reply` 给原始调用者；要么通过返回控制元组给 `gen_fsm` 模块返回 `Reply`。

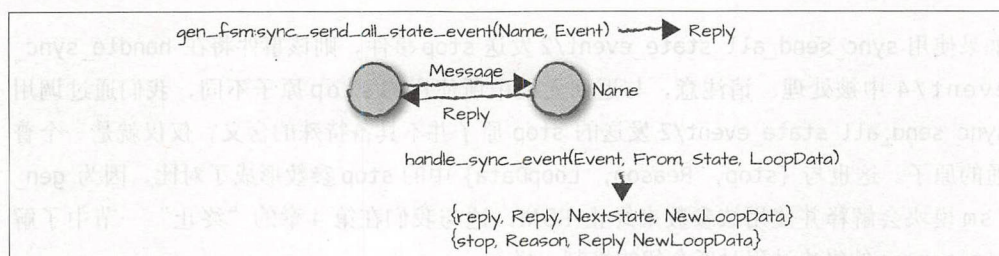


图 6-8: 同步式全状态事件

```
gen_fsm:sync_send_event(NameScope, Event) -> Reply
gen_fsm:sync_send_event(NameScope, Event, Timeout) -> Reply
```

139

```
gen_fsm:sync_send_all_state_event(NameScope, Event) -> Reply
gen_fsm:sync_send_all_state_event(NameScope, Event, Timeout) -> Reply
```

```
Mod:State/3,
Mod:handle_sync_event/4 -> {reply,Reply,NextState,NewLoopData}
                           {reply,Reply,NextState,NewLoopData,Timeout}
                           {reply,Reply,NextState,NewLoopData,hibernate}
                           {next_state,NextState,NewLoopData}
                           {next_state,NextState,NewLoopData,Timeout}
                           {next_state,NextState,NewLoopData,hibernate}
                           {stop,Reason,Reply,NewLoopData}
                           {stop,Reason,NewLoopData}
```

让我们使用 `sync_send_all_state_event/2` 函数来触发咖啡机的正常终止。毕竟，就停止操作而言，不论当前是什么状态都无所谓。

终止

我们的咖啡机可以有两种终止原因。它要么正常停止，要么非正常终止——比如使用 `exit BIF` 或发生运行时错误的时候。如果 FSM 调用 `process_flag(trap_exit, true)` 设置了捕获 `exit` 信号，则当发生异常终止时，会调用回调模块中的 `terminate/3`（参见图 6-9）。反之，如果 FSM 不捕获 `exit`，则 FSM 直接终止，其 `exit` 信号将传播到与其相关联的其他进程。

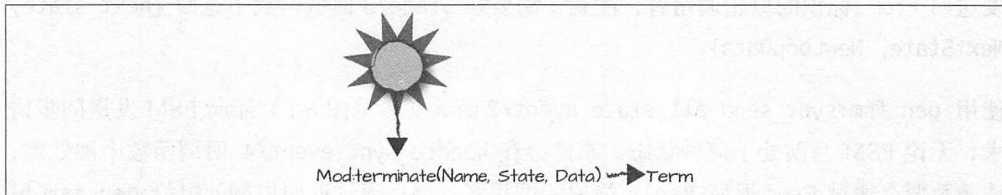


图 6-9: 终止

140 如果使用 `sync_send_all_state_event/2` 发送 `stop` 事件，则该事件将在 `handle_sync_event/4` 中被处理。请注意，与返回元组中所使用的 `stop` 原子不同，我们通过调用 `sync_send_all_state_event/2` 发送的 `stop` 原子并不具备特殊的含义，仅仅就是一个普通的原子。这也与 `{stop, Reason, LoopData}` 中的 `stop` 参数形成了对比，因为 `gen_fsm` 模块会解释并使用该参数来终止 FSM。这与我们在第 4 章的“终止”一节中了解 `gen_server` 的终止过程时所介绍的机制一样：

```
stop() -> gen_fsm:sync_send_all_state_event(?MODULE, stop).
```

```
handle_sync_event(stop, _From, _State, LoopData) ->
    {stop, normal, LoopData}.
```

```
terminate(_Reason, payment, {_Type, _Price, Paid}) ->
    hw:return_change(Paid);
terminate(_Reason, _StateName, _LoopData) ->
    ok.
```

还要注意，在 `terminate` 函数中，我们是如何针对各类状态执行不同清理动作的。如果客户支付了饮料费用，终止前他们必须收到退款。通过将此逻辑写在 `terminate/3` 中，即使发生异常终止资金也保证会退还。下面是一个例子：

```
1> {ok, Pid} = coffee_fsm:start_link().
Display:Make Your Selection
{ok,<0.35.0>}
2> coffee_fsm:americano().
```



```
Display:Please pay:150
ok
3> coffee_fsm:pay(100).
Display:Please pay:50
ok
4> exit(Pid, crash).
Display:Shutting Down
true
Machine:Returned 100 in change

=ERROR REPORT==== 3-Mar-2013::12:01:25 ===
** State machine coffee_fsm terminating
** Last message in was { 'EXIT' ,<0.33.0>,crash}
** When State == payment
**     Data == {americano,150,100}
** Reason for termination =
** crash
** exception exit: crash
```

总结

141

至此，我们已介绍了 gen_fsm 行为模式背后的原理。虽然它可能不是最常用的行为模式，但如果它恰巧符合你的应用程序的需要，那么将大大简化你的任务，使你的代码更易于阅读和维护。表 6-2 列出了本章涵盖的重要的函数。

表6-2: gen_fsm 回调

gen_fsm 函数 / 动作	gen_fsm 回调函数
gen_fsm:start/3, gen_fsm:start/4, gen_fsm:start_link/3, gen_fsm:start_link/4	Module:init/1
gen_fsm:send_event/2	Module:StateName/2
gen_fsm:send_all_state_event/2	Module:handle_event/3
gen_fsm:sync_send_event/2, gen_fsm:sync_send_event/3	Module:StateName/3
gen_fsm:sync_send_all_state_event/2, gen_fsm:sync_send_all_state_event/3	Module:handle_sync_event/4
Pid ! Msg 方式发送的消息、来自监视器的消息、exit 消息、来自端口和套接字的消息、来自节点监视器或其他非 OTP 类消息……	Module:handle_info/2
通过返回 {stop, ...} 触发终止，或者是非正常终止但是设定了捕捉 exit 的情况	Module:terminate/3

查看 `gen_fsm` 模块的手册页, 你可以在 `gen_fsm.erl` 源文件中找到该行为模式库的实现代码。如果你以前看过 `gen_server.erl` 的代码, 请特别注意这二者是如何使用 `gen.erl` 辅助模块的, 因为其他行为模式也使用该辅助模块。

亲力亲为

在进入下一章之前, 为什么不实现一个 FSM 来加深对其设计、编码和测试过程的认识呢? 如果你暂时还不打算编写代码, 请从第 8 章的示例中下载代码, 阅读它, 并进行试运行, 因为我们会在将来的示例中使用其中的控制器 (`controller`)。这个例子的有趣之处在于, 不同的行为模式实例各自代表了一部手机, 彼此对话。这是一个典型的大规模并发应用程序示例, 其中使用进程来表示和控制资源或设备。其中的手机利用归属位置寄存器 (`home location register`) 数据库, 把在网络上注册的用户映射到具有唯一性的电话号码上, 而且是利用我们在第 2 章“ETS: Erlang 元素存储”一节中实现的 `hlr` 模块完成的。

142 电话控制器

在我们的蜂窝系统中, 没有中央交换机。相反, 对于连接到网络中的每部手机, 我们创建一个电话控制器 (`phone controller`), 它会与其余控制器进行交互。每个控制器都实现为一个 FSM 进程, 其持有对应电话的状态。电话控制器之间的所有通信都必须是异步的, 以防止系统阻塞。完成以下 API 以实现 `phone_fsm.erl` 模块中的电话控制器。

```
start_link(PhoneNumber) -> {ok, FsmPid}.
```

为指定的电话号码启动新的电话控制器 FSM 进程, 该号码是与调用进程相链接的。这还会使电话控制器进程附着 (`attach`) 到其在归属位置寄存器 (HLR) 中的电话号码。

```
stop(FsmPid) -> ok.
```

停止 `FsmPid` 对应的电话控制器 FSM 进程, 并且将其与 HLR 中的电话号码相剥离。

```
connect(FsmPid) -> ok., disconnect(FsmPid) -> ok.
```

由电话调用, 用于将电话自身附着到电话控制器 FSM 进程。必须这么做, 电话控制器才知道要向哪里发送来电和去电相关信息的回应。调用 `connect` 函数通常是在手机启动时或连接到另一个 FSM 进程时发生。请注意, 我们是通过 `pid` 而非其电话号码连接到 FSM 进程。`disconnect` 函数用于将电话从手机控制器 FSM 进程中分离出来。

```
action(FsmPid, Action) -> ok.
```

从手机向 `FsmPid` 所指的手机控制器发送一个动作。有效的动作包括:

`{outbound, PhoneNumber}`

尝试连接到另一部电话。

`accept`

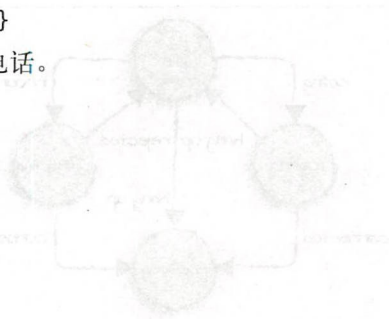
接听呼叫请求。

`reject`

拒绝呼叫请求。

`hangup`

挂断进行中的通话。



调用下述函数可以在交换机内的电话控制器之间发送事件：

143

`busy(FsmPid) -> ok.`

向 `FsmPid` 发送一个 `busy` 事件，一般作为对入站请求的回复，表示当前电话忙不能接受呼叫。

`reject(FsmPid) -> ok.`

向 `FsmPid` 发送 `reject` 事件，一般作为对入站请求的回复，表示拒绝该呼叫。

`accept(FsmPid) -> ok.`

向 `FsmPid` 发送 `accept` 事件，一般作为对入站请求的回复，表示我们接受该呼叫。

`hangup(FsmPid) -> ok.`

向 `FsmPid` 发送 `hangup` 事件以终止正在进行的呼叫。

`inbound(FsmPid) -> ok.`

向 `FsmPid` 发送 `inbound` 事件，请求建立呼叫。

结合所给的 API，图 6-10 显示了控制器 FSM 整体应该是什么样的。请注意，该 FSM 是不完整的：由于竞态条件的存在，事件的顺序可能错乱，或者由于网络 / 软件错误而导致事件丢失。在编码之前，请确保已经对它进行了审查，并添加了缺失的事件和状态转换。审查接口有助于你发现到底缺失了哪些事件和状态转换。

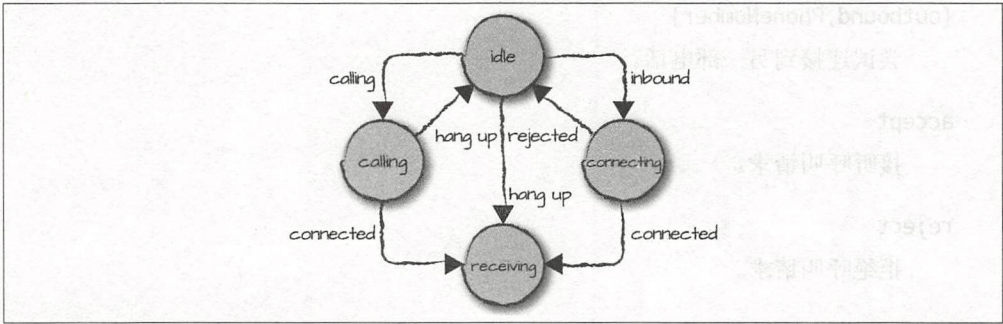


图 6-10: 电话控制器 FSM

让我们测试一下

每个电话控制器都连接到手机。你不必为手机编写代码,代码已在 *Phone.erl* 模块中提供,并具有以下 API。

144 `start_link(PhoneNumber) -> {ok, PhonePid}.`

为 *PhoneNumber* 启动一个新的电话,并与调用进程链接。

`stop(PhonePid) -> ok.`

停止 *PhonePid* 对应的电话。

`action(PhonePid, Action) -> ok.`

执行电话用户请求的针对 *PhonePid* 的动作。有效的动作包括:

`{call, PhoneNumber}`

呼叫 *PhoneNumber*。

`accept`

接受呼叫请求。

`reject`

拒绝呼叫请求。

`hangup`

挂断进行中的呼叫。

调用一个动作将导致事件被发送到电话的电话控制器上,使用的正是我们上一节中定义的电话控制器的 API。


```
reply(PhonePid, Reply) -> ok.
```

从电话控制器发送回应给电话。有效的回应事件包括：

```
{inbound, PhoneNumber}.
```

来自 PhoneNumber 的呼叫已抵达。

```
accept
```

呼出已接受。

```
invalid
```

呼出的号码无效。

```
reject
```

呼出被拒绝。

```
busy
```

呼出的电话忙。

```
hangup
```

呼出已挂断。

这些回应事件将导致电话进程在控制台的输出格式为 *PhonePid: PhoneNumber: Event* 的信息。例如：

```
<0,459,0>: 103618: hangup
```

你应该在不同的节点上运行 `hlr` 和电话控制器，并启动手机。最终的测试是让手机自己打电话并返回一个忙碌的信号。下面是一个试运行，包含三个手机：

```
1> hlr:new().
{ok,<0.34.0>}
2> phone_fsm:start_link("123").
{ok,<0.36.0>}
3> phone_fsm:start_link("124").
{ok,<0.38.0>}
4> phone_fsm:start_link("125").
{ok,<0.40.0>}
5> {ok,P123}=phone:start_link("123").
{ok,<0.42.0>}
6> {ok,P124}=phone:start_link("124").
{ok,<0.44.0>}
7> {ok,P125}=phone:start_link("125").
{ok,<0.46.0>}
```

```
8> phone:action(P123, {call,"124"}).
<0.44.0>: 124: inbound call from 123
ok
9> phone:action(P124, accept).
<0.42.0>: 123: call accepted
ok
10> phone:action(P125, {call,"123"}).
<0.46.0>: 125: busy
ok
11> phone:action(P125, {call,"124"}).
<0.46.0>: 125: busy
ok
```

接下来是什么

在下一章中，我们来看看另一类 worker 行为模式——`gen_event`（通用事件管理器）。与 `gen_server` 和 `gen_fsm` 略有不同，因为单个事件管理器实例允许同时关联多个回调模块。这些回调模块被称为 *handler*（处理程序），并且如果这些处理程序的实现足够通用的话，它们甚至可以跨多个不同的事件管理器重用。我们可以利用它们来了解基站控制器中发生的状况。

事件处理器

你们公司生产的移动设备频率管理服务器 (mobile frequency server) 在市场上反响极佳, 且变得愈发流行。为了能够对系统的性能和运行状态进行可视化, 你被要求实现一套监控软件, 不仅要能够收集重要流程的统计信息以及日志, 还要能够当出现问题时及时发出警告。于是问题就来了。如果你是在办公室里, 你会希望屏幕上有个小器件 (widget), 有问题时它会闪烁。如果你离开了办公桌, 你可能希望保留小器件功能的同时, 系统还能够在出问题给你自动发送邮件。而当你离开了办公室, 你更希望以短信 (SMS) 或者传呼 (pager message) 的方式收到提示, 而不是邮件。你的其他值班同事可能更希望接到电话, 而不是短信或者传呼, 因为这两种方式很难在半夜把他们唤醒。可见, 同样类型的事件必须能够在不同时刻触发不同的动作, 而具体触发何种动作则由一些外部因素决定。gen_event (通用事件处理器) 行为模式正是解决这一难题的好办法。

事件

事件实质上代表了系统状态的改变。事件可能源于 CPU 负载飙高、硬件故障或者端口活动产生的追踪信息等。事件管理器 (event manager) 是一种 Erlang 进程, 它能够接收各种类型的事件, 包括警报 (alarm)、警告 (warning)、设备状态改变、调试追踪、网络连接活动问题等。当这些事件生成后, 它们就被以消息的形式发送给管理器, 如图 7-1 所示。对于这样的每一条消息, 系统可能会希望对应地采取一组动作, 就如我们先前讨论过的那样: 生成 SNMP 陷阱 (trap); 发送邮件、短信、传呼; 收集统计数据; 打印消息到控制台; 或者记录日志到文件。我们把生成事件的一方叫作生产者 (producer), 而把接收事件并进行处理的一方叫作消费者 (consumer)。

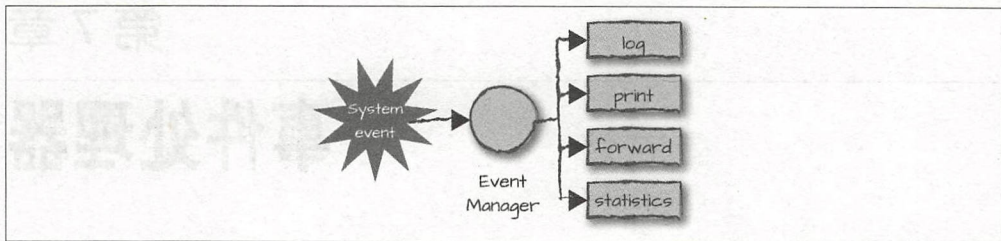


图 7-1: 事件管理器与处理器

148 而我们所说的事件处理器 (event handlers) 则是一些行为模式回调模块, 它们负责完成各种类型的动作。它们订阅了发给管理器的事件, 并且管理器允许多个不同的处理器 (handler) 订阅相同的事件。而多个处理不同事件的管理器又可以使用到相同的事件处理器。比如某个事件处理器允许你记录事件到文件, 另一个则允许你输出事件到控制台, 第三个则是收集统计信息, 那么你可以在事件管理器里同时把它们都用上, 便于处理调试追踪 (debug trace) 或者设备状态改变等各种消息。而且, 还可以在运行过程中动态添加、移除、查询、升级事件处理器, 这些功能都是在事件管理器代码中实现的。假如现在你打算自己写代码实现这样的事件管理与处理的功能, 那么考虑一下, 其中哪些是可以在 Erlang 系统中通用的, 哪些则是专用于你的应用的? 表 7-1 展示了分析结果。

表 7-1: 事件处理器与管理器中通用的与专用的代码

通用的	专用的
• 启动/停止事件管理器	• 事件
• 发送事件	• 事件处理器
• 发送同步请求	• 初始化事件处理器
• 转发事件/请求给处理器	• 事件处理器的循环数据
• 添加/删除处理器	• 处理事件/请求
• 升级处理器	• 清理

启动和停止事件管理器进程是通用的, 为它们注册别名也是。具体的进程名称和发给管理器的事件是专用的, 但是生产者发送消息、管理器接收消息以及调用消息处理器这些则是通用的。事件处理器本身是专用的, 为了使它能被初始化以及被移除时能够完成清理所做的那些也是专用的 (事件管理器停止时也会导致事件处理器被移除)。事件处理器如何处理事件是专用的, 它们的循环数据也是。最后, 升级事件处理器是通用的, 但是具体各个事件处理器升级时如何对状态进行处理则是专用的。

149 让我们看看 `gen_event` 行为模式模块。虽说 `gen_event` 行为模式也是基于 `gen_server` 建

立的，但是 `gen_event` 行为模式和我们已经见到过的其他行为模式还是有很大不同的。

通用事件管理器 / 处理器

通用事件管理器 / 处理器是标准库应用（standard library application）的一部分，并且和其他所有行为模式一样，代码也是分成通用的（generic）和专用的（specific）两部分。所有的通用代码包含在 `gen_event` 模块里。运行这一部分代码的进程通常被我们称为“事件管理器”（event manager）。而通过一组回调函数订阅和处理事件的回调模块则被我们称为“事件处理器”（event handler）。每个事件处理器负责完成一个特定的、事件驱动的任务，其中的代码属于专用的。其他行为模式通常是每个实例中只允许指定一个回调模块，但是事件管理器却不同，它可以接受零个或多个事件处理器，如图 7-2 所示。尽管可能存在多个事件处理器，但是它们都运行在一个事件管理器进程内。

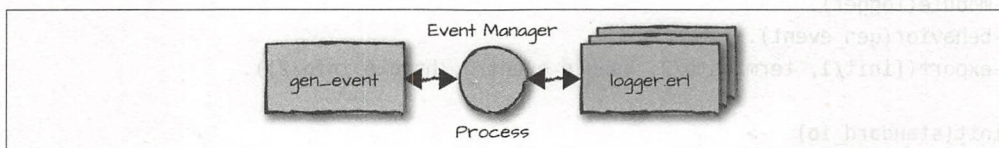


图 7-2: 处理器回调模块

启动 / 停止事件管理器

`gen_event:start_link(NameScope)` 函数用于启动新的事件管理器。通过 `NameScope` 可以指定进程的本地（local）/ 全局（global）名称，或者是 `via` 模块——在第 4 章的“全局化”一节里我们解释过。如果你不想注册进程，那就使用 `start_link/0` 然后通过 `pid` 与其通信。和其他行为模式不同，`start_link/0` 不接受回调模块、参数或选项。它也不会调用任何回调函数。管理器所做的仅仅是把事件管理器列表设为一个空列表：

```
gen_event:start()
gen_event:start(NameScope)
gen_event:start_link()
gen_event:start_link(NameScope) -> {ok,Pid}
                                   {error,{already_started,Pid}}
gen_event:stop(NameScope) -> ok
```

因为你并不会调用 `init/1` 回调函数，因而也就不可能在 `init/1` 函数中返回 `stop` 或者 `ignore`，或者产生运行错误等问题，这样一来就基本不可能碰到什么错误了，除了事件管理器想要注册的名字已经被注册过了这种情况。

调用 `gen_event:stop/1` 可以停止事件管理器。

150 添加事件处理器

我们已经能够启动和停止事件管理器了，接下来实现并添加一个事件处理器。在事件管理器进程运行过程中，我们可以动态添加和移除事件处理器。由于你可以实现那种能运行于不同事件管理器中处理多种类型事件的事件处理器，所以从某种程度来说，事件处理器这种行为模式比其他行为模式还要更通用。

在我们的 `logger` 例子里，实现了一个事件处理器，它能够记录事件和非预期的消息 (`unexpected messages`) 到标准输入/输出或者文件（具体采用何种方式取决于将其添加到事件管理器时所提供的参数）。和我们介绍过的其他通用行为模式一样，也是从 `behavior` 指令开始，然后导出回调函数：

```
-module(logger).  
-behavior(gen_event).  
-export([init/1, terminate/2, handle_event/2, handle_info/2]).
```

```
init(standard_io) ->  
    {ok, {standard_io, 1}};  
init({file, File}) ->  
    {ok, Fd} = file:open(File, write),  
    {ok, {Fd, 1}};  
init(Args) ->  
    {error, {args, Args}}.
```

如果我们调用 `gen_event:add_handler(Name, Mod, Args)` 函数，在 `Mod` 中实现的事件处理器就被添加到了事件管理器中。事件管理器调用 `Mod:init(Args)` 回调函数，而该函数返回 `{ok, LoopData}`，其中 `LoopData` 是针对那个事件处理器的。在我们的例子里，循环数据里有一个元组，这个元组由一个文件描述符（或原子 `standard_io`）和一个整数 1 构成，该整数起计数作用——每次收到一个事件，就把它加 1。如果我们传递的参数是原子 `standard_io`，则所有的事件都会被输出到 `shell` 中。传递 `{file, File}` 作为参数，其中 `file` 是原子，而 `File` 则是一个包含了文件名的字符串，其将会把所有日志记录到那个文件中。

为了能够管理多个事件，事件管理器把它的事件处理器，以及这些事件处理器的循环数据存储到了列表中。图 7-3 展示的是我们的事件处理器实例以及它的循环数据被事件管理器添加到列表中的过程，这一列表里还存储着其他的事件处理器以及它们的循环数据。

151 你不但可以在一个事件管理器里添加多个事件处理器，而且可以把同一个事件处理器重复添加多次，每次添加将对应不同的循环数据。在我们的场景里，我们要添加两个日志

事件处理器，其中一个负责保存事件信息到文件，另一个则负责输出到 shell。另一种做法是把 Mod 参数指定为 {Module, Id}，其中 Id 可以是任意的 Erlang 项。如果 Id 是唯一的，那么同一个事件管理器中的基于相同回调模块的多个事件处理器就能够相互区分。

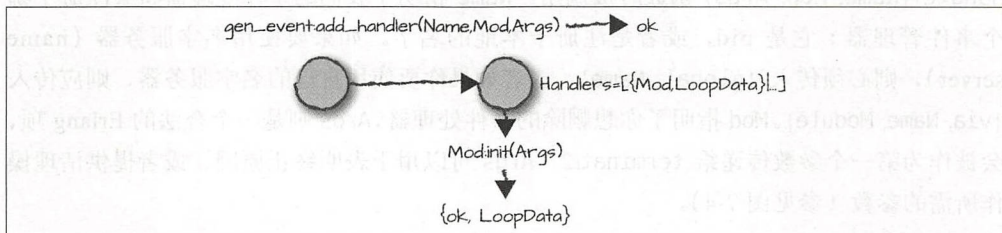


图 7-3: 添加处理器

```
gen_event:add_handler(NameScope, Mod, Args) -> {'EXIT', Reason}
```

```
ok
Term
```

```
Mod:init/1 -> {ok, LoopData}
             {ok, LoopData, hibernate}
             Term
```

添加不存在的事件处理器将会导致事件管理器调用 `Mod:init/1` 失败，并返回 `{'EXIT', Reason}`，其中 `Reason` 是运行时错误原因，这种情况下为 `undef`，意为函数未定义。如果 `init/1` 回调函数中的任何表达式失败，也会导致返回 `{'EXIT', Reason}`。要明确的是，`{'EXIT', Reason}` 是由 `try-catch` 表达式捕获来的，而不是异常（exception）。

不管 `init/1` 回调返回的数据值是什么——即使不是 `{ok, LoopData}`——该数据值依然会被返回。其中初学者常犯的一种错误是只返回原子 `ok` 而未包含 `LoopData`。然而只要 `init/1` 返回的不是 `{ok, LoopData}`，该事件处理器就不会被添加到事件管理器里。所以只返回 `ok` 而未包含 `LoopData` 是行不通的，事件处理器不会被添加。

在我们的例子里，如果事件处理器启动时，前两个分句都未能匹配，则 `init/1` 会返回 `{error, {args, Args}}`，并且事件管理器不会把它添加到事件处理器列表中。因此，尽管 `init/1` 可以返回任意值，但只应当返回格式如 `{ok, LoopData}` 和 `{error, Reason}` 的值，以避免混乱。

和其他行为模式一样，你也可以让你的事件管理器在没有事件时（即两个事件之间）休眠。只要其中一个事件处理器返回 `hibernate` 就能触发这一行为。使用休眠要小心，只在事件间隔长时使用。因为把进程休眠这一操作会使进程在休眠前和唤醒后，触发全量（full-sweep）垃圾回收。在短时间内收到大量事件，这种行为方式不会是你想要的。

删除事件处理器

现在我们已经添加了事件处理器，再来看看如要删除它该如何做。logger 回调模块导出了 `terminate(Args, LoopData)` 回调函数。该函数会在调用 `gen_event:delete_handler(Name, Mod, Args)` 函数时被调用。`Name` 指明了我们的事件处理器将会注册于哪个事件管理器；它是 pid，或者是注册于本地的名字。如果要使用名字服务器 (name server)，则必须传入 `{global, Name}`，或者如果你要使用自己的名字服务器，则应传入 `{via, Name, Module}`。`Mod` 指明了你想删除的事件处理器，`Args` 则是一个合法的 Erlang 项，会被作为第一个参数传递给 `terminate/2`。`Args` 可以用于表明终止原因，或者提供清理操作所需的参数（参见图 7-4）。

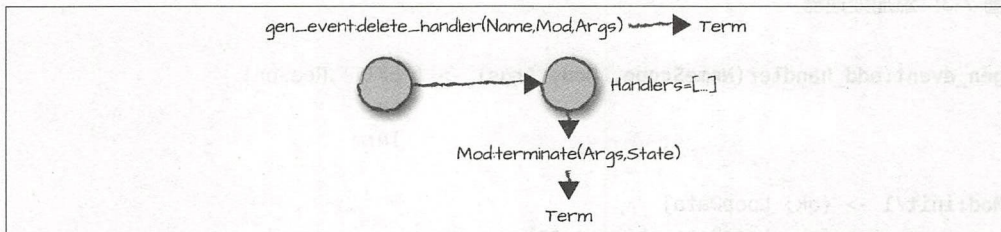


图7-4：删除处理器

在我们的例子里，如果要移除 logger 事件处理器，则必须满足打印日志到标准 I/O 或者文件这两种情况之一：

```
terminate(_Reason, {standard_io, Count}) ->
    {count, Count};
terminate(_Reason, {Fd, Count}) ->
    file:close(Fd),
    {count, Count}.
```

153 > 当 `terminate/2` 函数返回后，事件处理器也就从调用 `delete_handler/3` 时 `Name` 指定的事件管理器进程中移除了。其他也添加了同样事件处理器的事件管理器则不会受影响。如果以同样的 `Mod` 注册了多个事件处理器，例如一个是用于记录日志到 `standard_io`，而另一个是记录日志到文件，它们将会被以添加顺序的相反顺序删除。如果你使用 `gen_event:stop/1` 停止事件管理器，则所有的事件处理器都会以原因 `stop` 被移除。

注意 `terminate/2` 返回的 `Term`，它成了 `delete_handler/3` 调用的返回值。在我们的例子里，返回的是日志记录的累计次数 `{count, Count}`，这样我们就能够知道在该事件处理器终止前有多少事件流过了该事件处理器。但是当要升级 (`upgrade`) 事件处理器时，返回的 `Term` 应该是完整的循环数据。我们会在本章后面介绍升级 (`upgrade`) 方面的内容。

尝试移除一个并没有注册过的事件处理器将导致返回值为 `{error, module_not_found}`。向并不存在的事件管理器添加和移除事件处理器——不管是 `pid` 还是以注册别名引用的该事件管理器——都会导致发起调用的进程终止，终止原因为 `noproc`。

```
gen_event:delete_handler(NameScope, Mod, Args) -> {error,module_not_found}
                                                {'EXIT',Reason}
                                                Term
```

```
Mod:terminate/2 -> Term
```

发送同步的或异步的事件

事件发送给事件管理器后会被转给事件处理器，这一过程可以是同步的或者异步的，选择何种方式取决于是否需要控制生产者产生事件的速率。事件是被事件管理器受理的，然后各个事件处理器会被依次序逐个调用。如果你发送许多事件给事件管理器，而处理这些事件的事件处理器中有几个很慢，那么就像第 15 章中“同步调用与异步调用”一节描述的那样，你的消息队列可能会增长而吞吐量下降，可见不要让你的事件处理器成为瓶颈。我们会在第 15 章的“平衡你的系统”一节讨论处理大量消息的技术。

`gen_event:notify/2` 函数发送异步的事件给所有的事件处理器后立刻返回 `ok`。每个事件处理器的 `Mod:handle_event/2` 函数会被依次逐个调用，当然这里指的是那些已经被添加到事件管理器中的事件处理器。`gen_event:sync_notify/2` 同样也会使所有事件处理器的 `Mod:handle_event/2` 回调函数被调用。这和异步方式（即 `gen_event:notify/2`）的区别在于，只有当所有事件处理器的回调都被调用后才会返回 `ok`。

考虑一下对于 `logger` 来说，我们应该如何实现它的 `handle_event/2` 回调函数：

```
handle_event(Event, {Fd, Count}) ->
    print(Fd, Count, Event, "Event"),
    {ok, {Fd, Count+1}}.

print(Fd, Count, Event, Tag) ->
    io:format(Fd, "Id:~w Time:~w Date:~w~n"++Tag++":~w~n",
              [Count,time(),date(),Event]).
```

如图 7-5 所示，`handle_event/2` 回调接收两个参数，第一个参数是事件本身，第二个参数要么是原子 `standard_io`，要么是一个（在 `init/1` 回调里打开的）文件描述符。`print/4` 函数调用 `io:format/3` 函数来输出计数值、当前日期时间、事件标签以及事件本身。

◀ 154

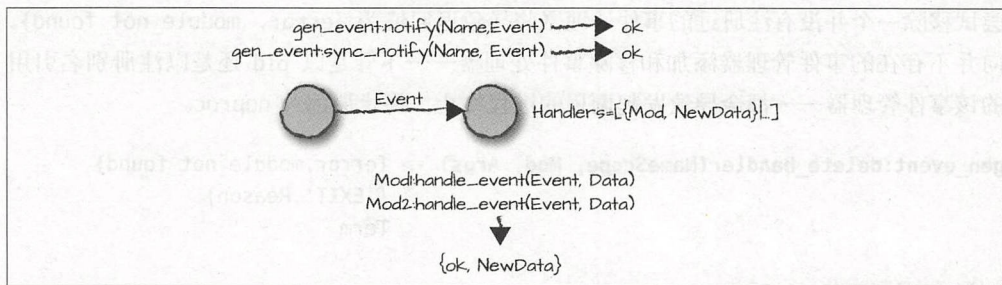


图7-5: 通知

如果我们的事件处理器收到任何非 OTP 兼容的事件，不管是来自链接 (link)、捕获的 exit 信号、进程监视器、监视的分布式 Erlang 节点，或者来自 Pid ! Msg 方式，它们都会由事件处理器的 handle_info/2 回调函数来处理。

```

handle_info(Event, {Fd, Count}) ->
    print(Fd, Count, Event, "Unknown"),
    {ok, {Fd, Count+1}}.
  
```

logger 中 handle_info/2 的实现几乎和 handle_event/2 一样，差别只在于前者传递给 print 函数的标签 (tag) 值是 Unknown——表明事件来源未知。

```

gen_event:notify(NameScope, Event)
gen_event:sync_notify(Name, Event) -> ok
  
```

```

Mod:handle_event(Event, Data)
Mod:handle_info(Event, Data) -> {ok, NewData}
                                {ok, NewData, hibernate}
                                remove_handler
                                {swap_handler, Args1, NewData, Handler2, Args2}
  
```

如果事件处理器的 handle_event/2 或者 handle_info/2 函数返回 remove_handler，则 Mod:terminate(remove_handler, Data) 会被调用并且该事件处理器被移除。在本章后面我们会讨论交换 (swapping) 事件处理器方面的问题。在此之前，让我们先确保我们在事件处理器里写的那些代码能够正常运作。

在 shell 命令 1 里，我们直接启动了事件管理器，没有注册，没有链接到父进程。这样即使 shell 进程崩溃，事件管理器也不会受到影响。然后我们添加了一个事件处理器，并发送了两个事件，一个同步的一个异步的：

```

155 1> {ok, P} = gen_event:start().
      {ok, <0.35.0>}
      2> gen_event:add_handler(P, logger, {file, "alarmlog"}).
  
```



```

ok
3> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
ok
4> gen_event:sync_notify(P, {clear_alarm, no_frequency}).
ok

```

注意两次都返回了原子 `ok`。但它们在语义上不同，`shell` 命令 4 是直到所有事件处理器都执行后才返回 `ok`。

在 `shell` 命令 5 中，我们添加了第二个事件处理器实例，这一次我们把事件直接输出到标准 I/O。在 `shell` 命令 6 中，我们发送了一条非 OTP 兼容的消息，然后两个事件处理器实例的 `handle_info/2` 回调函数把这一消息记录下并输出到了 `shell` 里：

```

5> gen_event:add_handler(P, logger, standard_io).
ok
6> P ! sending_junk.
Id:1 Time:{18,59,25} Date:{2013,4,26}
Unknown:sending_junk
sending_junk

```

在 `shell` 命令 7 和 8 里，我们读取了 `alarmlog` 文件的二进制内容并把它们输出到了 `shell` 里。可以看到我们先前分别以同步和异步方式发送的两个事件，以及被 `handle_info/2` 接收到的未知（unknown）消息。

```

7> {ok, Binary} = file:read_file("alarmlog").
{ok,<<"Id:1 Time:{18,59,10} Date:{2013,4,26}\nEvent:{set_alarm,{no_frequency,...
8> io:format(Binary).
Id:1 Time:{18,59,10} Date:{2013,4,26}
Event:{set_alarm,{no_frequency,<0.32.0>}}
Id:2 Time:{18,59,14} Date:{2013,4,26}
Event:{clear_alarm,no_frequency}
Id:3 Time:{18,59,25} Date:{2013,4,26}
Unknown:sending_junk
ok
9> gen_event:delete_handler(P, freq_overload, stop).
{error,module_not_found}
10> gen_event:stop(P).
ok

```

然后我们打算移除 `freq_overload`——一个并不存在于这一事件管理器里的事件处理器。和预期的一样，返回了错误 `module_not_found`。最后，我们停止了事件管理器，这样做默认会关闭所有事件处理器。

你可以从本书的代码仓库中下载 `logger` 事件处理器的代码 (<https://github.com/>)

francescoc/scalabilitywitherlangotp), 并运行一下。测试在事件管理器已停止 (或崩溃) 的情况下向其发送同步和异步消息, 然后使用 `start_link` 启动它, 再让 shell 崩溃。最后, 试着指出如果添加事件处理器时提供无效的文件名会发生什么。

获取数据

让我们来实现另一个事件处理器, 它能够存储指标 (metrics)。每次记录一个事件, 我们还会把 ETS 表中用于告诉我们该事件已经被记录了多少次的累计值加 1。如果这是该事件第一次出现, 我们就在表中创建一个新条目。看看代码, 如果需要的话, 也看看 `ets` 模块的参考手册:

```
-module(counters).  
-behavior(gen_event).  
-export([init/1, terminate/2, handle_event/2, handle_info/2]).  
-export([get_counters/1, handle_call/2]).
```

```
get_counters(Pid) ->  
    gen_event:call(Pid, counters, get_counters).
```

```
init(_) ->  
    TableId = ets:new(counters, []),  
    {ok, TableId}.
```

```
terminate(_Reason, TableId) ->  
    Counters = ets:tab2list(TableId),  
    ets:delete(TableId),  
    {counters, Counters}.
```

```
handle_event(Event, TableId) ->  
    try ets:update_counter(TableId, Event, 1) of  
        _ok -> {ok, TableId}  
    catch  
        error:_ -> ets:insert(TableId, {Event, 1}),  
                {ok, TableId}  
    end.
```

```
handle_call(get_counters, TableId) ->  
    {ok, {counters, ets:tab2list(TableId)}, TableId}.
```

```
handle_info(_, TableId) ->  
    {ok, TableId}.
```

在这个例子里我们感兴趣的主要是累计值 (counters) 是怎么获取的。使用 `gen_event:sync_event/2` 是不行的, 虽说它是同步的, 但是它把事件传递给各个事件处理器

后就返回 ok 了。而且我们还需要能够指明想把同步消息传递给哪个事件处理器，所以我们使用 `gen_event:call(NameScope, Mod, Message)` 函数。

如图 7-6 所示，事件处理器在 `Mod:handle_call/2` 回调函数中同步地收到请求并且返回了格式为 `{ok, Reply, NewData}` 的元组，其中 Reply 是请求的返回值。

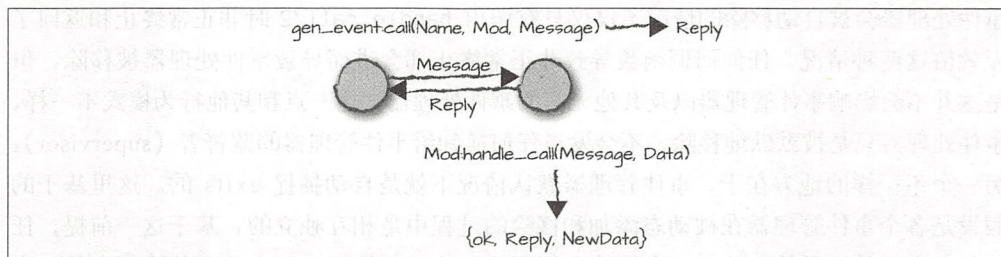


图7-6: 调用

```

gen_event:call(NameScope, Mod, Request)
gen_event:call(NameScope, Mod, Request, Timeout) -> Reply
                                                    {error, bad_module}
                                                    {error, {'EXIT', Reason}}
                                                    {error, Term}

```

```

Mod:handle_call(Event, Data) -> {ok, Reply, NewData}
                                Term

```

`gen_event:call/3` 默认的超时值是 5000 毫秒。可以通过传入以毫秒为单位的整数值 `Timeout` 来改变这一设定，也可以传入原子 `infinity`。如果 `Mod` 不是一个已经添加到 `NameScope` 中的事件处理器，则 `{error, bad_module}` 被返回。如果回调函数 `handle_call/2` 在处理请求时非正常终止，则会返回 `{error, {'EXIT', Reason}}`。最后，如果 `handle_call/2` 返回的不是 `{ok, Reply, NewData}`，则 `gen_event:call` 的返回值将会是 `{error, Term}`。不管是这些错误情形的哪一种，事件处理器都会被从事件管理器中移除，这一过程不会影响到其他事件处理器。

```

1> {ok, P} = gen_event:start().
{ok,<0.35.0>}
2> gen_event:add_handler(P, counters, {}).
ok
3> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
ok
4> gen_event:notify(P, {event, {frequency_denied, self()}}).
ok
5> gen_event:notify(P, {event, {frequency_denied, self()}}).

```

ok

```
6> counters:get_counters(P).
{counters,[{{event},{frequency_denied,<0.33.0>}},2},
          {{set_alarm,{no_frequency,<0.33.0>}},1}]}
```

158 对错误以及无效返回值的处理

事件处理器会被自动移除的情况不仅仅只有调用 `handle_call/2` 时非正常终止和返回了无效值这两种情况。任何回调函数导致非正常终止都会进而导致事件处理器被移除。但是这并不会影响事件管理器以及其他无辜的事件处理器。这一点和其他行为模式不一样，事件处理器只是被默默地移除，不会发送任何通知给事件管理器的监督者（supervisor）。另一个不一样的地方在于，事件管理器默认情况下就是自动捕捉 `exits` 的。这里基于的假设是各个事件管理器在被动态添加和移除的过程中是相互独立的，基于这一前提，任何一个事件管理器的崩溃都不会影响到周围的环境（参见图 7-7）。然而故障隔离是一个很好的特性，但是在发生故障时保持沉默却不是。

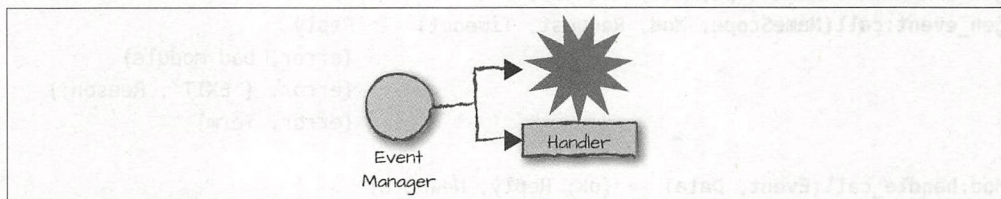


图7-7：处理器崩溃

为了更好地理解事件处理器非正常终止时发生的事，让我们使用下面的代码片段作为例子：

```
-module(crash_example).
-behavior(gen_event).
-export([init/1, terminate/2, handle_event/2]).
```

```
init(normal) -> {ok, []};
init(return) -> error;
init(ok)      -> ok;
init(crash)   -> exit(crash).
```

```
terminate(_Reason, _LoopData) -> ok.
```

```
handle_event(crash, _LoopData) -> 1/0;
handle_event(return, _LoopData) -> error.
```

在添加这一事件处理器到事件管理器时，根据传入的参数不同，以及在运行过程中，我们发送的消息不同，可以人为制造运行时错误或者令其返回无效的值。逐步执行下面的

shell 命令，其中的不同请求对应的正是不同的错误情形：

```
1> {ok,P}=gen_event:start().
{ok,<0.35.0>}
2> gen_event:which_handlers(P).
[]
3> gen_event:add_handler(P, crash_example, return).
error
4> gen_event:which_handlers(P).
[]
5> gen_event:add_handler(P, crash_example, normal).
ok
6> gen_event:which_handlers(P).
[crash_example]
7> gen_event:notify(P, crash).
ok
=ERROR REPORT==== 27-Apr-2013::09:27:49 ===
** gen_event handler crash_example crashed.
** Was installed in <0.35.0>
** Last event was: crash
** When handler state == []
** Reason == {badarith,
               [{crash_example,handle_event,2,
                  [{file,"crash_example.erl"},{line,13}]}],
               ...]}
8> gen_event:which_handlers(P).
[]
9> gen_event:add_handler(P, crash_example, normal).
ok
10> gen_event:notify(P, return).
ok
=ERROR REPORT==== 27-Apr-2013::09:28:41 ===
** gen_event handler crash_example crashed.
** Was installed in <0.35.0>
** Last event was: return
** When handler state == []
** Reason == error
11> gen_event:which_handlers(P).
[]
```

虽然有错误报告产生（这些报告的内容会在第 9 章的“SASL application”小节中进行介绍），但是并没有运行时错误产生，因此，也没有 EXIT 信号产生。发送事件只是默默地出错，事件处理器被移除的这一行为没有任何进程或人员能够获悉。

要绕过这个问题，可以调用 `gen_event:add_sup_handler/3` 函数把事件处理器和发起该

调用的进程连接 (connect) 起来。其工作方式与 `add_handler/3` 相同, 但额外的副作用是使得发起调用的进程开始监视对应的事件处理器。如果发生异常或者回调函数在处理事件时返回了不正确的值, 则一条格式为 `{gen_event_EXIT, Mod, Reason}` 的消息会被发送给添加该事件处理器的进程。其中 `Reason` 可以为下述之一。

- `normal`: 如果回调函数返回 `remove_handler` 或者处理器被使用 `delete_handler/3` 移除。
- `shutdown`: 如果事件管理器被其 supervisor 停止, 或者被 `stop/1` 调用停止。
- `{EXIT, Term}`: 如果发生了运行时错误。
- `Term`: 如果回调返回了任何 `{ok, LoopData}` 或者 `{ok, Reply, LoopData}` 之外的东西。
- `{swapped, NewMod, Pid}`: 其中的 `Pid` 指代的进程交换了当前处理器。

我们将在下一节讨论交换 (swapping) 事件处理器。

监视是双向的。如果添加该事件处理器的进程终止了, 则该事件处理器会被以 `{stop, Reason}` 作为原因移除。这种做法能够确保同一处理器的多个实例不会被意外地反复加入管理器, 比如某个行为模式由于被反复重启而导致处理程序被重复添加。



出错务必大声!

如果你正在编写具有高可用性和容错能力要求的系统, 则你最不想看到的就是由于在 `init/1` 中出错, 导致处理器完全没被添加甚至默默被删除。所以请始终坚持检查 `add_handler/3` 和 `add_sup_handler/3` 调用的返回值。如果你必须使用 `add_handler`, 则请确保所有可能存在 bug, 或者依赖于外部资源 (如磁盘, 因而可能遇到空间不足导致的错误等), 或者可能遇到错乱数据的代码, 把它们全都放在 `try-catch` 表达式之中。如果可能, 最好使用 `add_sup_handler/3`, 并对其返回值进行模式匹配, 以确保处理程序已正确添加, 并注意收到的一切异常消息。你一定不希望你的报警系统在发生故障时不会发出任何报警!

交换事件处理器

事件管理器提供了相应的功能可以用于在运行时交换 (swap) 事件处理器。这一功能允许把当前事件处理器的状态传递给新的事件处理器, 保证进程中的事件不会出现丢失。调用 `gen_event:swap_handler/3` 时的第二个参数是一个元组, 其中包含了我们想要替换的事件处理器的回调模块名, 以及将会传递给其 `terminate` 函数的参数。第三个参数是一个元组, 指明了新事件处理器的回调模块名, 以及传递给其 `init` 函数的参数。图 7-8 展示了这些参数, 以及交换事件处理器时的步骤。

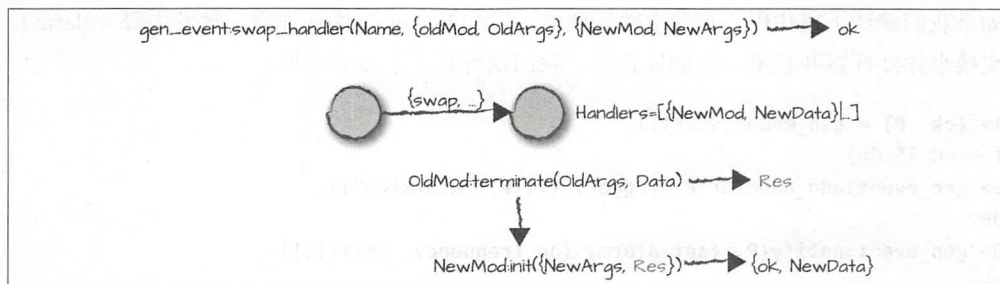


图7-8: 交换处理器

旧事件处理器的 `terminate` 回调函数首先被调用。它的返回值 `Res` 和另一个参数（也是要传递给 `init` 函数的）一起放在一个元组中传递给新事件处理器的 `init` 函数。简直是不能再简单了！如果你想在交换事件处理器的同时启动监督，就使用 `gen_event:swap_sup_handler/3`。你正要换掉的事件处理器此时不要求已经处于被监督状态。

◀ 161

解释交换过程的最佳办法莫过于举一个例子。让我们扩展一下 `logger` 事件处理器，使其能够在两种模式——记录到文件和记录到标准 I/O——之间切换（flip）。我们扩展了 `terminate` 函数以处理原因为 `swap` 时的情况，并在该情况下返回 `Res`——一个格式为 `{Type, Count}` 的元组。`Type` 要么是文件描述符，要么是原子 `standard_io`，而 `Count` 则代表了要记录的下一项的唯一编号（unique ID）。因为我们不清楚交换后的 `logger` 想如何处理事件，所以我们没有关闭文件，而是让交换后的 `logger` 在 `init/1` 函数中处理。

在 `init/1` 调用中，我们添加了两处理情况，在添加事件处理器时，它们接收的 `Args` 是相同的，并且是来自 `terminate` 返回的结果。因此，如果我们原本是记录到文件而现在想交换为记录到标准 I/O，就得关闭文件并返回 `{ok, {standard_io, Count}}`。如果我们原本是输出到标准 I/O，则可以打开文件并把事件写入其中。在这两种情况下，我们都会承接 `Count` 的当前值：

```

init({standard_io, {Fd, Count}}) when is_pid(Fd) ->
    file:close(Fd),
    {ok, {standard_io, Count}};
init({File, {standard_io, Count}}) when is_list(File) ->
    {ok, Fd} = file:open(File, write),
    {ok, {Fd, Count}};
...

terminate(swap, {Type, Count}) ->
    {Type, Count};
...

```

162 如果我们测试上述代码，启动事件管理器、添加 logger 事件处理器、触发告警（alarm）、交换事件处理器并且第二次触发告警，我们就得到了下面的结果：

```
1> {ok, P} = gen_event:start().
{ok,<0.35.0>}
2> gen_event:add_handler(P, logger, {file, "alarmlog"}).
ok
3> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
ok
4> gen_event:swap_handler(P, {logger, swap}, {logger, standard_io}).
ok
5> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
Id:2 Time:{10,1,16} Date:{2013,4,27}
Event:{set_alarm,{no_frequency,<0.33.0>}}
ok
6> {ok, Binary}=file:read_file("alarmlog"), io:format(Binary).
Id:1 Time:{10,1,16} Date:{2013,4,27}
Event:{set_alarm,{no_frequency,<0.33.0>}}
ok
```

融会贯通

现在我们有了一个事件处理器，让我们把它包装到模块里，把事件管理器的 API 隐藏并提供更加直观化和与应用紧密相关的函数集。我们把这些都汇聚在 `freq_overload` 模块里，它负责启动事件管理器，并提供 API 来设置和清除 `no_frequency` 警报，并在 client 拒绝频率时生成事件。它还为添加和移除事件处理器的函数提供了一层包装。我们把和事件处理器相关的函数调用——如获取累计值（counters）、把输出到文件切换为输出到标准 I/O——局限在了内部：

```
-module(freq_overload).
-export([start_link/0, add/2, delete/2]).
-export([no_frequency/0, frequency_available/0, frequency_denied/0]).

start_link() ->
case gen_event:start_link({local, ?MODULE}) of
  {ok, Pid} ->
    add(counters, {}),
    add(logger, {file, "log"}),
    {ok, Pid};
  Error ->
    Error
end.
```



```

no_frequency() ->
    gen_event:notify(?MODULE, {set_alarm, {no_frequency, self()}}).
frequency_available() ->
    gen_event:notify(?MODULE, {clear_alarm, no_frequency}).
frequency_denied() ->
    gen_event:notify(?MODULE, {event, {frequency_denied, self()}}).

```

```

add(M,A) -> gen_event:add_sup_handler(?MODULE, M, A).
delete(M,A) -> gen_event:delete_handler(?MODULE, M, A).

```

注意我们是如何在调用 `freq_overload:start_link/0` 的过程中添加计数器 (counters) 的。这种做法可以确保, 即使事件管理器重启, counters 和 logger 事件处理器依然会被自动添加。缺点是我们无法监督事件管理器里的事件处理器, 因此当它们崩溃时我们无法得知。如果你想要另一个进程来监视事件处理器, 就使用 `freq_overload:add/2`, 因为其中用的是 `gen_event:add_sup_handler/3`。

当启动警报并发出事件时, 我们也把频率分配器的 pid 包含了进来。这使得我们可以区分不同的分配器 (明确警报来源或者事件发起者), 这样运维人员就能够确定是哪些服务器过载 (overutilized), 是否需要为它们分配一个更大的频率池。我们希望每当分配器把频率用尽时都触发警报, 而当它又有空闲频率时则清除警报。如果 client 恰好分配走了最后一个频率, 我们就调用 `freq_overload:no_frequency/0`, 触发 `no_frequency` 警报。如果频率被释放回来的一刻, 恰巧正处于无频率可用的状态, 则我们通过调用 `freq_overload:frequency_available/0` 清除警报。我们还会在用户每次请求分配频率被拒时调用 `freq_overload:frequency_denied/0` 函数。之所以要把这一情况划分为一种单独的事件, 是因为我们可能会仅仅处于频率耗尽但是未拒绝过任何请求的状态。这些添加到 `frequency.erl` 里的代码简明而直白:

```

allocate({[], Allocated}, _Pid) ->
    freq_overload:frequency_denied(),
    {[[], Allocated}, {error, no_frequency}};
allocate({[Res|Resources], Allocated}, Pid) ->
    case Resources of
        [] -> freq_overload:no_frequency();
        _ -> ok
    end,
    {[Resources, [[Res, Pid]|Allocated]], {ok, Res}}.

deallocate({Free, Allocated}, Res) ->
    case Free of
        [] -> freq_overload:frequency_available();
        _ -> ok
    end,

```

```
NewAllocated = lists:keydelete(Res, 1, Allocated),
[[Res|Free], NewAllocated}.
```

现在我们已经改好了频率分配器里的其他代码，同时也实现好了自己的 `freq_overload` 事件管理器，让我们把 `logger` 和 `counters` 这两个事件处理器也添加到事件管理器吧，这样它们将一同运行，如图 7-9 所示。在触发警报的同时，我们还记录下了警报事件本身。

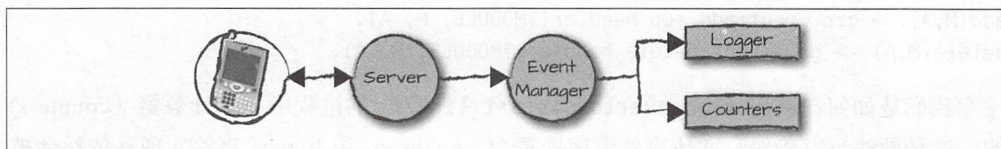


图 7-9: 处理器示例

164 我们启动频率服务器以及事件管理器并添加了第二个日志事件处理器，这个日志事件处理器会把日志输出到 `shell` 里。在我们的例子中，频率分配器总共拥有 6 个频率。在 `shell` 命令 4 中，我们把这些频率都分配光了，触发了 `no_frequency` 警报。虽然触发了警报，但最后一次请求本身是成功的，返回了 `{ok, 15}`：

```
1> frequency:start_link().
{ok,<0.35.0>}
2> freq_overload:start_link().
{ok,<0.37.0>}
3> freq_overload:add(logger, standard_io).
ok
4> frequency:allocate(), frequency:allocate(), frequency:allocate(),
    frequency:allocate(), frequency:allocate(), frequency:allocate().
Id:1 Time:{10,41,25} Date:{2015,2,28}
Event:{set_alarm,{no_frequency,<0.35.0>}}
{ok,15}
5> frequency:allocate().
Id:2 Time:{10,41,46} Date:{2015,2,28}
Event:{event,{frequency_denied,<0.35.0>}}
{error,no_frequency}
6> frequency:allocate().
Id:3 Time:{10,42,0} Date:{2015,2,28}
Event:{event,{frequency_denied,<0.35.0>}}
{error,no_frequency}
7> frequency:deallocate(15).
Id:4 Time:{10,42,16} Date:{2015,2,28}
Event:{clear_alarm,no_frequency}
ok
8> counters:get_counters(freq_overload).
```



```
{counters, [{set_alarm, {no_frequency, <0.35.0>}, 1},
             {clear_alarm, no_frequency}, 1},
            {event, {frequency_denied, <0.35.0>}, 2}]}
```

在警报已经触发的情况下，我们又尝试了两次频率分配，都失败了。当我们在 shell 命令 7 中释放了一个频率后，警报解除了。当我们获取 counters 时，看到频率拒绝分配发生了两次，而 no_frequency 警报触发和解除了各一次。

SASL 警报处理器

165

我们在本章提到过警报处理器 (alarm handler)，但并没有给出一个合适的定义，现在是时候澄清一下了。所谓警报处理器是系统中的一个部件，它负责记录出现的问题，并执行相应的动作。如果你的系统消耗的内存量已经达到一定量，或者已经耗尽了磁盘空间（或者频率），你会想要触发警报。当内存使用量降低或者老的日志文件被删除后，相应的警报就被解除。在任何一个时刻，应当能够查阅当前活跃的警报列表，并获取当前状况 (issue) 的一份快照。

SASL 警报处理器进程实质是 Erlang 运行时系统自带的一套事件管理器和事件处理器。这是一套很基础的警报处理器，我们鼓励你把它用于自己的项目时，如果有必要可以进行功能上的替换和完善。开发 Erlang 系统的哲学是：由简而始，迭代递增。SASL 警报处理器同样遵循这一哲学。

根据你安装 Erlang 的方式不同，SASL 警报处理器很可能已经是启动了的。在你的 shell 里运行 `whereis(alarm_handler)` 可以确定这一点。如果你获得的是原子 `undefined`，就在 shell 里输入 `application:start(sasl)` 启动警报处理器。你可能会看到一些进度报告被输出到 shell 里，具体内容和你安装 Erlang 的方式也有关系。我们将分别在第 9 章、第 11 章和第 16 章分别介绍报告、警报方面更一般性的内容，以及其他 SASL 中包含的有用工具。此刻，不必在意这些报告。

如果 `whereis/1` 返回了一个 pid，则说明警报处理器已经运行，你不需要做什么额外的事情，除了尝试一下：

```
1> whereis(alarm_handler).
<0.41.0>
2> alarm_handler:set_alarm({103, fan_failure}).

=INFO REPORT==== 26-Apr-2013::08:23:27 ===
    alarm_handler: {set,{103,fan_failure}}
ok
3> alarm_handler:set_alarm({104, cabinet_door_open}).
```

```

=INFO REPORT==== 26-Apr-2013::08:23:43 ===
    alarm_handler: {set,{104,cabinet_door_open}}
ok
4> alarm_handler:clear_alarm(104).

=INFO REPORT==== 26-Apr-2013::08:24:04 ===
    alarm_handler: {clear,104}
ok
5> alarm_handler:get_alarms().
[{103,fan_failure}]

```

166 在我们的例子中，呈现的场景是机架上的冷却风扇出故障了。一名系统管理员走到机架前，打开柜门检查发生的状况，关闭柜门，转身回去订购替换用的风扇。我们所做的是触发了两个警报，编号分别为 103 和 104。这些编号在清除警报时会用到，shell 命令 4 里当柜门关闭时执行的正是这样的操作。这些导出的函数是对 SASL 事件管理器和事件处理器的包装：

```

alarm_handler:set_alarm({AlarmId, Description}) -> ok
alarm_handler:clear_alarm(AlarmId) -> ok
alarm_handler:get_alarms() -> [{AlarmId, Description}]

```

在复杂的系统中，你可能会面对数百种不同的警报，它们各自有不同的严重等级划分，而且清除某一种警报默认情况下还会清除一堆与它相关联的其他警报。你会希望能够保证统计精确，一切事情都被记录，甚至在更高级的系统中还会运行一些能够立刻自动执行动作的代理（agent）。举例来说，在风扇出故障这种情况下，你可能会想自动关闭该机柜中全部的设备，以避免过热。但现有的事件处理器没有任何一个具备此功能，而且可伸缩性也达不到要求。但在开始着手改进前，请记住，开发 Erlang 系统较好的方式是不断迭代设计、开发和测试过程。

要想替换和完善现存的事件处理器其实很容易。你需要处理 {set_alarm, {AlarmId, AlarmDescr}} 和 {clear_alarm, AlarmId} 事件。如果你想替换现有的事件处理器，使用 swap_handler/3。

```

gen_event:swap_handler(alarm_handler,
                        {alarm_handler, swap}, {NewHandler, Args})

```

你的新处理器的 init 函数应当匹配模式 {Args, {alarm_handler, Alarms}}，其中 Args 是在 swap_handler/3 调用时传入的，而 {alarm_handler, Alarms} 则是从老的事件处理器的 terminate/2 函数返回的。Alarms 是一个列表，其中每个元素都是形如 {AlarmId, Description} 的元组。

总结

本章我们介绍了事件管理器是如何处理事件的。现在你应该已经很好地理解了 `gen_event` 行为模式的优势，不会再去自己造一个重复的，或者以复杂的方式在自己的子系统中整合一个类似的功能。在事件管理器和其他 OTP 行为模式之间最大的差别在于，它是一种一对多（one-to-many）的关系，你可以把许多事件处理器都关联到同一个事件管理器上。我们介绍的最重要的功能和回调函数都列到了表 7-2 里。

表 7-2: `gen_event` 回调

gen_event 函数或动作	gen_event 回调函数
gen_event:start/0, gen_event:start/1, gen_event:start_link/0, gen_event:start_link/1	Module:init/1
gen_event:add_handler/3, gen_event:add_sup_handler/3	
gen_event:swap_handler/3,	Module1:terminate/2,
gen_event:swap_sup_handler/3	Module2:init/1
gen_event:notify/2, gen_event:sync_notify/2	Module:handle_event/2
gen_event:call/3, gen_event:call/4	Module:handle_call/2
gen_event:delete_handler/3	Module:terminate/2
gen_event:stop/1	Module:terminate/2
Pid ! Msg 方式发送的消息、来自监视器的消息、exit 消息、来自端口和套接字的消息、来自节点监视器或其他非 OTP 类消息……	Module:handle_info/2

在继续阅读后面的内容之前，确保你已经温习了 `gen_event` 模块的手册页。其中的 `alarm_handler` 模块方面的内容是对本章内容的有益补充与完善。阅读这些代码后你会注意到，开发人员把用于启动和停止事件管理器的 `client` 函数和事件处理器本身的函数整合到了一起。

接下来是什么

事件管理器是我们介绍的最后一个 worker 行为模式。`gen_event`、`gen_server` 以及 `gen_fsm` 以及那些你自己写的行为模式都是在监督树（supervision tree）中启动的。下一章我们将介绍监督者（supervisor）行为模式，它负责启动、停止和监视其他 supervisor 以及 worker。我们会在第 10 章介绍如何编写自己的行为模式。当我们在第 16 章讲到监视（monitoring）和抢救性（preemptive）支持时会更进一步地介绍警报（alarm）对你的系统实现高可用性和高可靠性的重要性。

监督者

现在我们已经能够监视和处理各种可预见的错误了（比如频率不足），接下来需要把那些由于数据错乱或代码有 bug 导致的意外错误控制住。难点在于，在意外错误出现之前我们根本不知道要发生什么，这与频率分配器返回给客户端的错误或事件管理器提供的警报之类的可预期的错误不同。我们可以推理、猜测，并尝试添加代码来处理意外错误，并祈祷能够如愿。使用基于属性（property-based）的自动测试生成工具（如 QuickCheck 或 PropEr）绝对可以帮助你创建一些自己永远不会想到的故障场景。然而除非你有超能力，否则你永远无法在故障发生前预测每一个可能的意外错误，更不用说处理它。

为了应对错误或损坏的数据，开发人员常常在代码中实现自己的错误处理和恢复策略，结果增加了代码的复杂性以及维护成本，并且还只是处理了可能出现的问题中的一小部分，甚至最终已有的问题没解决几个，反而引入了更多的问题。毕竟，如果你根本不知道 bug 是什么你又怎么能处理呢？你有没有碰到过这样的 C 程序员，他检查了 `printf` 语句的返回值，但却不确定如果真的出现错误，到底该怎么办？如果你是从另一种支持异常处理的语言（如 Java 或 C++）转到 Erlang 的，回想一下你看到过多少除了一句 *TODO* 注释（意在提醒开发团队在将来某个时刻改正这一疏漏，然而实际上永远没有那一天）外什么也没有的 `catch` 表达式？

这就是通用监督者行为模式（generic supervisor behavior）登场的动力。它为开发人员承担了意外错误的处理和恢复责任。该行为以确定的和一致的方式处理监控、重启策略、竞态条件和边界情况等，其中许多细节是绝大多数开发人员考虑不到的。于是在拥有全面的错误恢复策略的同时，我们可以把 worker 行为写得更简单。现在就来看看监督者行为模式（supervisor behavior）的工作机制。

监督树

监督者指的是一类以监督和管理子进程作为其唯一任务的进程。它们分裂出一些进程并将它们与自己相链接 (link)。监督者通过捕捉退出 (trapping exits) 并接收 EXIT 信号, 使得当出现意外状况时, 可以采取适当的应对措施。措施多种多样, 从重启子进程到不启动, 到终止部分或者全部的与监督者链接的子进程, 甚至终止监督者自身等。子进程既可以是普通的工作进程, 也可以是其他监督者进程。

容错是通过创建监督树来实现的, 其中监督者是节点, 而工作者是树叶 (参见图 8-1)。各级别的监督者负责监视并处理自己启动的子树中的子进程。

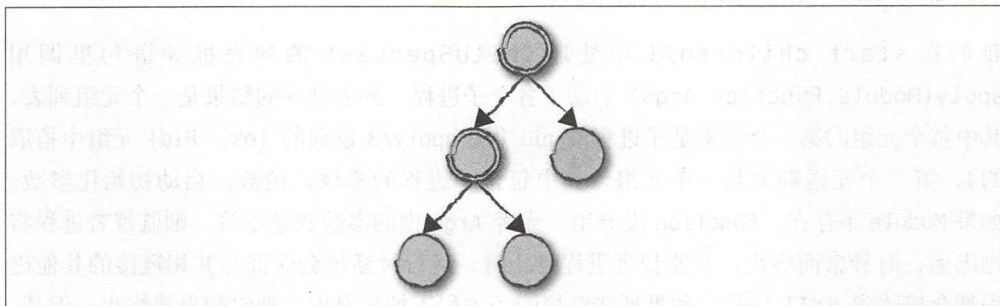


图 8-1: 监督树

图 8-1 中使用双环表示捕捉退出的进程。虽然在我们的例子中只有监督者捕捉退出 (trapping exits), 但实际上如果需要工作者也可以这么做, 没有限制。

让我们通过动手编写一个简单版的监督者开始学习吧。这将使我们能够在深入了解 OTP 监督者实现之前先了解其幕后需要执行的任务。只要提供一份子进程规格 (specification) 列表, 我们的简单版监督者就会按照要求启动子进程, 并将与它们链接。如果任何一个子进程异常终止, 简单版监督者将立即将其重启。否则, 如果子进程正常终止, 则从监督树中删除, 不再执行其他动作。停止监督者将导致其所有子进程被无条件终止。

以下是启动监督者和子进程的代码:

171

```
-module(my_supervisor).  
-export([start/2, init/1, stop/1]).
```

```
start(Name, ChildSpecList) ->  
    register(Name, Pid = spawn(?MODULE, init, [ChildSpecList])),  
    {ok, Pid}.
```



```
stop(Name) -> Name ! stop.
```

```
init(ChildSpecList) ->  
  process_flag(trap_exit, true),  
  loop(start_children(ChildSpecList)).
```

```
start_children(ChildSpecList) ->  
  [{element(2, apply(M,F,A)), {M,F,A}} || {M,F,A} <- ChildSpecList].
```

当启动 `my_supervisor` 时，我们向 `init/1` 函数传递了子进程规格。这是一个 `{Module, Function, Arguments}` 元组构成的列表，其中包含的函数将会在分裂时将子进程与其父进程相链接。我们假设这个函数总是返回 `{ok, Pid}`，其中 `Pid` 是新分裂的子进程的 ID。返回其他任何值都被视为启动出错。

我们在 `start_children/1` 中处理 `ChildSpecList` 的列表推导语句里调用 `apply(Module,Function,Args)` 启动了各个子进程。列表推导的结果是一个元组列表，其中每个元组的第一个元素是子进程的 `pid`（从 `apply/3` 返回的 `{ok, Pid}` 元组中拾取的），第二个元素则又是一个元组，其中包含子进程的模块、函数、启动初始化参数。如果 `Module` 不存在，`Function` 没导出，或者 `Args` 中的参数数量不符，则监督者进程将抛出运行时异常而终止。当监督者进程终止时，运行时系统会保证与其相链接的其他进程都会接收到 `EXIT` 信号。如果那些链接的子进程不捕捉退出，则它们也将终止。但是，它们如果捕捉退出，则需要处理 `EXIT` 信号，最有可能的方式是通过终止自身，从而将 `EXIT` 信号传播到其他相链接的进程。

一个有效的假设是——启动系统时不应该出现意外状况。基于这一假设，如果监督者无法正确启动一个子进程，它将终止其所有子进程并终止启动过程。虽说我们的系统是有弹性的，能够从错误中恢复，但是启动阶段就出错是不可接受的。

```
loop(ChildList) ->  
  receive  
    {'EXIT', Pid, normal} ->  
      loop(lists:keydelete(Pid,1,ChildList));  
    {'EXIT', Pid, _Reason} ->  
      NewChildList = restart_child(Pid, ChildList),  
      loop(NewChildList);  
  stop ->  
    terminate(ChildList)  
end.
```

```
restart_child(Pid, ChildList) ->  
  {Pid, {M,F,A}} = lists:keyfind(Pid, 1, ChildList),  
  {ok, NewPid} = apply(M,F,A),
```




```
lists:keyreplace(Pid,1,ChildList,{NewPid, {M,F,A}}).

terminate(ChildList) ->
  lists:foreach(fun({Pid, _}) -> exit(Pid, kill) end, ChildList).
```

监督者紧接着便开始循环，循环过程用到了从 `start_children/1` 返回的格式为 `{Pid, {Module, Function, Argument}}` 的元组列表。该元组列表其实就是监督者的状态。当子进程异常终止时，我们会使用到此状态中的信息，根据 `pid` 查找到该进程的启动函数，如有必要就使用它进行重启。如果想为监督者进程注册一个别名，那么可以通过 `name` 参数传递。之所以不在模块中直接硬编码别名，是因为在 Erlang 节点中经常会需要相同监督者的多个实例。

完成了所有子进程的启动后，监督者进程开始了“接收 - 求值”循环（`receive-evaluate loop`）。请注意，这与第3章“进程的骨架”一节中描述的进程骨架完全相同，并且与服务器、FSM 和事件处理程序进程中循环过程也相似。与我们在 Erlang 中已实现过的其他行为的唯一区别在于，此处我们只处理 `EXIT` 消息，并在收到 `stop` 消息时执行特定的操作。

在我们的监督者中，如果子进程是以 `normal` 原因终止的，则将该进程从 `ChildSpecList` 中删除，并且监督者继续监视其他子进程。如果是以 `normal` 以外的原因终止的，则子进程会被重新启动，并且其旧的 `pid` 将会被替换为子进程规格列表中元组 `{Pid, {Module, Function, Argument}}` 里的 `NewPid`。如果我们的监督者收到 `stop` 消息，它将遍历其子进程列表，终止每个进程。

让我们把 `my_supervisor` 用在先前写的 `coffee FSM` 上看看吧。在这样做之前，请别忘了编译 `coffee_fsm.erl` 和 `hw.erl`。不过，转念一想，还是不要编译 `hw.erl` 了。用监督者启动 `coffee FSM`，看看在 `hw.erl` 桩模块不可用的情况下会发生什么。当你看到输出的错误报告后，从 `shell` 里编译或加载 `hw.erl` 使其可访问：

```
1> my_supervisor:start(coffee_sup, [{coffee_fsm, start_link, []}]).
{ok, <0.39.0>}
```

```
=ERROR REPORT==== 4-May-2013::08:26:51 ===
Error in process <0.468.0> with exit value:
{undef, [{hw,reboot,[],[]},{coffee,init,0,[...]}]}
```

```
...<snip>...
```

```
=ERROR REPORT==== 4-May-2013::08:26:58 ===
Error in process <0.474.0> with exit value:
{undef, [{hw,reboot,[],[]},{coffee,init,0,[...]}]}
```

173



```

2> c(hw).
Machine:Rebooted Hardware
Display:Make Your Selection
{ok,hw}
3> Pid = whereis(coffee_fsm).
<0.476.0>
4> exit(Pid, kill).
Machine:Rebooted Hardware
Display:Make Your Selection
true
5> whereis(coffee).
<0.479.0>
6> my_supervisor:stop(coffee_sup).
stop
7> whereis(coffee).
undefined

```

发生了什么？coffee FSM 在其初始化函数中调用了 `hw:reboot/0`，导致了 `undef` 错误，原因是该模块无法加载。监督者收到 `EXIT` 信号并重启了该 FSM。但这会陷入循环重启，因为重启的 FSM 依然无法解决问题；它将继续崩溃，除非模块成功被加载并可用。在 shell 命令 2 中编译 `hw.erl` 模块，这同时也使其被加载，于是 coffee FSM 初始化成功并正确启动了，循环重启得以结束。

循环重启发生的条件是，进程由于某些原因导致异常终止，当对该进程进行重启时，由于继续产生同样的异常，于是进程再次崩溃并重新启动。监督者行为提供了相应的机制可以消除循环重启。我们会在本章稍后讨论。现在，回到我们的例子中。

在 shell 命令 3 中，我们找到了 FSM 的 pid，并使用它发送了一条退出信号，使得 coffee FSM 终止。它立即就被重启，从 `init/0` 函数在 shell 中产生的输出可以看出这一点。我们在 shell 命令 6 中停止了监督者，于是工作者也终止了。

现在该思考几个我们对其他行为模式也思考过的问题了。看看 `my_supervisor.erl` 中的代码，在查看表 8-1 中的答案之前，问问自己：哪些部分是通用的，哪些部分是专用的？^{注1}

174 表 8-1: 监督者中通用的代码与专用的代码

通用的	专用的
<ul style="list-style-type: none"> • 分裂监督者 • 启动子进程 	<ul style="list-style-type: none"> • 启动什么样的子进程 • 处理特定子进程的：

注1 如果你是一个读脚注的人，这是一件好事，因为你现在了解到了这是一个棘手的问题。



通用的	专用的
• 监视子进程	— 启动、重启
• 重启子进程	— 子进程依赖
• 停止监督者	• 监督者名称
• 清理	• 监督者行为

分裂监督者进程并注册是通用的，不管监督者要启动和监视的是什么样的子进程。监视子进程并重启它们也是通用的，同样，停止监督者并终止所有的子进程也是。换句话说，`my_supervisor.erl` 中的所有代码都是通用的。专用的部分则在传入的变量里。包括 `ChildSpecList`（子进程必须按照其中的顺序启动）以及监督者进程要注册的别名。

虽然我们的 `my_supervisor` 在某些状况下是适用的，但它作为监督者所实现的功能还相当浅显。我们决定不再继续增加例子的复杂度了，不过还是可以考虑添加一些参数的。我们之前看到了因为子进程启动失败而导致的无休止重试。监督者应该支持设置指定时间间隔内重启次数的上限，而不是无休止地重试，这样如果子进程未能正确启动，可以采取进一步的行动。依赖关系呢？如果子进程终止，难道监督者不应该提供终止和重启依赖该子进程的其他子进程的选项？这些都是 OTP 监督者行为库模块中囊括的一些配置参数，下面我们就来介绍。

OTP 监督者

在 OTP 中，我们的程序是由一个或多个监督树组成的。我们将本质上相似或具有依赖关系的工作者组合在一起，放在相同的子树下，按照依赖顺序启动它们。当绘制监督树图时，工作者行为通常被以圆圈表示，而监督者则以正方形表示。图 8-2 展示的是我们之前研究的频率分配器例子中监督结构的一种可能的样子。

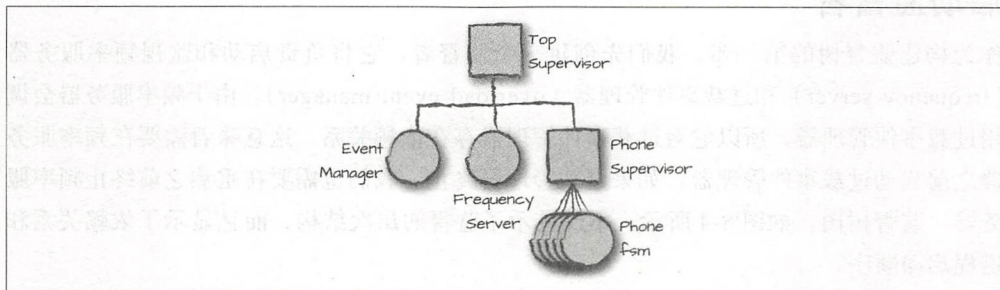


图 8-2: 监督树



175 考虑到依赖关系，顶级监督者首先启动的是负责处理警报（alarm）的事件管理器工作者（event manager worker）进程，因为它不依赖于其他工作者进程。然后，顶级监督者启动频率分配器（frequency allocator）进程，因为它向事件管理器发送警报。同级别最后一个被启动的进程是电话监督者（phone supervisor），负责启动和监视所有代表手机的 FSM。

请注意，我们是把相依赖的进程放在树中的一个子集合里，而把其他相关的进程放到了同样的树中的另一个子集合里，然后按照依赖顺序从左至右排列，依次启动。这部分作为系统中的监督策略，有时不是由开发人员来决定的（他们主要关注具体的工作者需要完成的事），而是由架构师基于对系统的大局，即各个组件间的交互情况来决定的。

监督者行为模式

在 OTP 中，监督者行为模式是在监督者库模块中实现的。与所有行为模式一样，非通用部分的代码写在回调模块里，其中含有 behavior 和 version 指令。监督者回调模块（supervisor callback module）需要导出一个回调函数，用于配置和启动由该监督者处理的树的子集（参见图 8-3）。

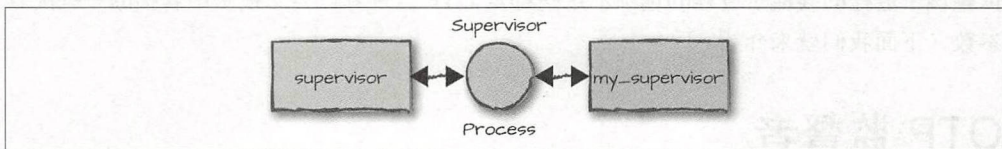


图 8-3: 通用监督者

你可能已经猜到此导出函数是 `init/1`，其中包含了监督者的配置。回调模块通常还提供用于启动监督者本身的函数。让我们更仔细地看看这些调用。

176 启动监督者

作为构建监督树的第一步，我们先创建一个监督者，它将负责启动和监视频率服务器（frequency server）和过载事件管理器（overload event manager）。由于频率服务器会调用过载事件管理器，所以它对过载事件管理器存在依赖关系。这意味着需要在频率服务器之前启动过载事件管理器，如果过载管理器终止，我们也需要在重启之前终止频率服务器。监督树图，如图 8-4 所示，不仅显示了监督的层次结构，而且显示了依赖关系和进程启动顺序。



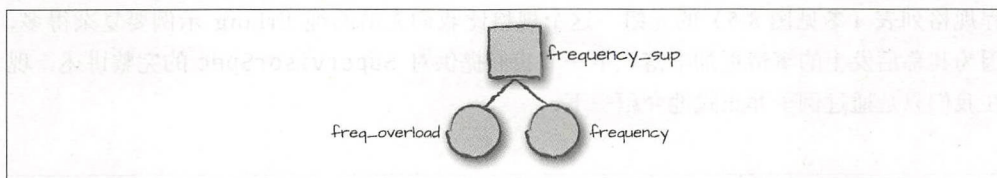


图 8-4: 频率分配服务器监督树

我们来看看频率监督者回调模块的代码。像所有其他行为模式一样，必须包括 `behavior` 指令。你可以使用 `start` 或 `start_link` 函数启动监督者，传入监督者名称（可选）、回调模块，以及用于传递给 `init/1` 的参数。与事件管理器一样，此处没有提供让你设置跟踪、日志记录或内存微调选项的 `Options` 参数：

```

-module(frequency_sup).
-behavior(supervisor).

-export([start_link/0, init/1]).
-export([stop/0]).

start_link() ->
    supervisor:start_link({local,?MODULE},?MODULE, []).

stop() ->
    exit(whereis(?MODULE), shutdown).

init(_) ->
    ChildSpecList = [child(freq_overload), child(frequency)],
    {ok,{rest_for_one, 2, 3600}, ChildSpecList}.

child(Module) ->
    {Module, {Module, start_link, []},
     permanent, 2000, worker, [Module]}.
  
```

在我们的示例中，`start_link/3` 调用中的 `[]` 表示的是发送到 `init/1` 回调的参数，而不是 `Options`。你无法在监督者启动时设置 `sys` 选项，但可以在监督者启动后设置。与其他行为的另一个区别是，监督者没有向开发人员暴露内置的停止功能。因为通常它们是由它们的监督者终止，或者随节点本身终止。对于那些不想写永不停止的系统，并且坚持想从 `shell` 关闭监督者的人，看看我们包括的 `stop/0` 函数，它模拟了更高级监督者的关闭过程。

调用 `start_link/3` 导致调用 `init/1` 回调函数。该函数返回一个格式为 `{ok, SupervisorSpec}` 的元组，其中 `SupervisorSpec` 是一个包含了监督者配置参数以及子进

177



程规格列表（参见图 8-5）的元组。这个规格比我们先前的纯 Erlang 示例要复杂得多，因为其幕后发生的事情更加丰富。下一节我们提供对 SupervisorSpec 的完整讲述。现在我们只是通过例子非正式地介绍一下。

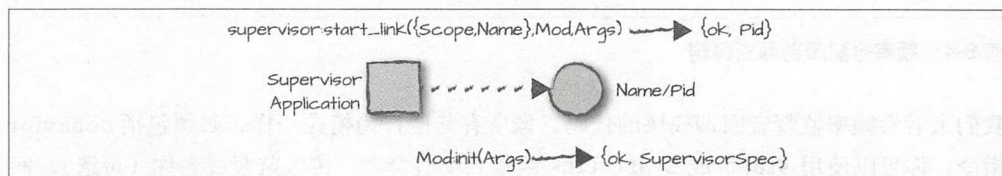


图 8-5: 通用监督者

在我们的示例中，SupervisorSpec 配置参数元组的第一个元素告诉监督者，如果一个子进程终止，所有在它之前启动的子进程也要终止，然后再重启它们。一般来说，我们把这个元素称为重启策略（restart strategy），而且为了获得此情景想要的效果，我们指定的是 rest_for_one 策略。紧随 rest_for_one 策略之后，元组中的数字 2 和 3600 分别表示强度（intensity）和周期（period），指明监督者每小时（3600 秒）最多允许发生两次子进程异常终止。如果超过这个数字，监督者将终止自身及其子进程，并发送一个原因为 shutdown 的退出信号给所有与其相链接的进程。因此，如果这个监督者是某个更大的监督树的一部分，则监督该监督者的上级监督者会收到退出信号并采取适当的行动。

SupervisorSpec 配置参数元组中的第二个元素是子进程规格列表。列表中的每一项都是一个元组，指明了启动和管理静态子进程的详细信息。在我们的示例中，元组中的第一个元素是一个（相对于其监督者范围内而言的）唯一标识符。之后是 {Module,Function,Arguments} 元组，它指明了工作者的启动函数，预期返回 {ok,Pid}，启动后将会被与监督者相链接。接下来，我们找到重启指令——原子 permanent，它指定了当监督者重启工作者时，总是应该重启此工作者。

178 重启指令之后是关机指令，这里是 2000。它告诉监督者当发送 EXIT 信号之后，给予 2000 毫秒的等待时间，子进程将在这一时间内关闭（包括 terminate 函数所花费的时间）。terminate 不保证一定会被调用，原因是子进程可能正忙于服务其他请求，甚至永远不会取出邮箱中收到的 EXIT 信号。

之后，原子 worker 表示子进程的性质是工作者而非监督者，最后是一个单元素列表 [Module]，指定的是实现该工作者的回调模块。

```
supervisor:start_link(NameScope, Mod, Args)
supervisor:start_link(Mod, Args) -> {ok, Pid}
                                   {error, Error}
```



ignore

```
Mod:init/1 -> {ok,{RestartStrategy,MaxR,MaxT},{ChildSpecList}}
            ignore
```

因为记住 SupervisorSpec 的所有字段的用途及其顺序很难，所以 Erlang 18.0 和之后的版本允许以映射（map）方式指定这些参数。这里是另一种 init/1 和 child/1 的实现，它们返回的 SupervisorSpec 是映射而非元组：

```
init(_) ->
    ChildSpecList = [child(overload), child(frequency)],
    SupFlags = #{strategy => rest_for_one,
                 intensity => 2, period => 3600},
    {ok, {SupFlags, ChildSpecList}}.
```

```
child(Module) ->
    #{id => Module,
      start => {Module, start_link, []},
      restart => permanent,
      shutdown => 2000,
      type => worker,
      modules => [Module]}.
```

正如你所看到的，把 SupervisorSpec 以映射表达使得代码更容易阅读，因为相较于元组，映射中各个字段都有对应的名称。如果你使用的是 Erlang 18.0 或后续版本，请使用映射作为你的监督者子进程规格。

监督者与其他所有行为一样，可以使用注册名或 pid 来引用。注册监督者的时候，{local,Name} 和 {global,Name} 都是合法的 NameScope 值。你还可以使用 {via,Module,Name} 元组中所指的名称注册表，其中 Module 模块导出了与全局名称注册表中定义的相同的 API。

init/1 回调函数通常返回整个元组，包含了重启配置和子进程规格列表。但是它还可以返回 ignore，这时监督者会以 normal 为原因终止。你有没有注意到，监督者没有导出 start/2,3 函数，目的是强制你链接到父级进程。在下一节中，我们将详细介绍所有可用的选项并更详细地介绍重启策略。我们把这些选项和策略统称为监督者规格（supervisor specification）。

179

监督者规格

监督者规格是一个包含了两个元素的元组（参见图 8-6）：

- 与该监督者紧密相关的（非通用的）重启策略。
- 该监督者负责的所有静态 worker 相关的子进程规格。

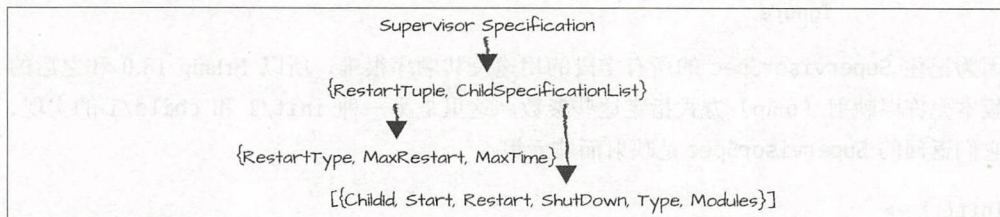


图 8-6: 监督者规格

我们来更仔细地看看这些值，先从重启元组开始。

重启规格

重启元组的格式如下：

```
{RestartType, MaxRestart, MaxTime}
```

它指明了如果某个子进程异常终止，应当如何对待监督树中的其他子进程。我们所谓的“子进程”指的可能是工作者进程或另一个监督者进程。从 Erlang 18.0 开始，还可以使用映射来指定这些信息。该映射定义重启规格时的类型定义如下：

```
#{strategy => strategy(),
  intensity => non_neg_integer(),
  period => pos_integer()}
```

有 4 种不同的重启类型：`one_for_one`、`one_for_all`、`rest_for_one` 和 `simple_one_for_one`。在 `one_for_one` 策略下（参见图 8-7），只重启崩溃了的进程。如果工作者进程相互之间无依赖关系，终止一个进程不会影响其他进程，则这个策略很合适。想象一个监督者，它监督着大量的工作者进程，影响到数十万用户的即时消息会话。如果这些进程中的任何一个崩溃，将仅影响到该崩溃进程控制的用户会话，而其他所有的工作者进程可以继续独立运行。

使用 `one_for_all` 策略时，如图 8-8 所示，如果进程终止，则所有进程也都被终止并重启。此策略一般用在全部或大部分进程彼此依赖时。想象一个用于处理协议栈的非常复杂的 FSM。为了简化设计，整个 FSM 被拆分成一个个小的 FSM，彼此异步通信，相互依赖。如果任何一个终止了，其他的也必须被终止。对于类似这样的情况，选择 `one_for_all` 策略。

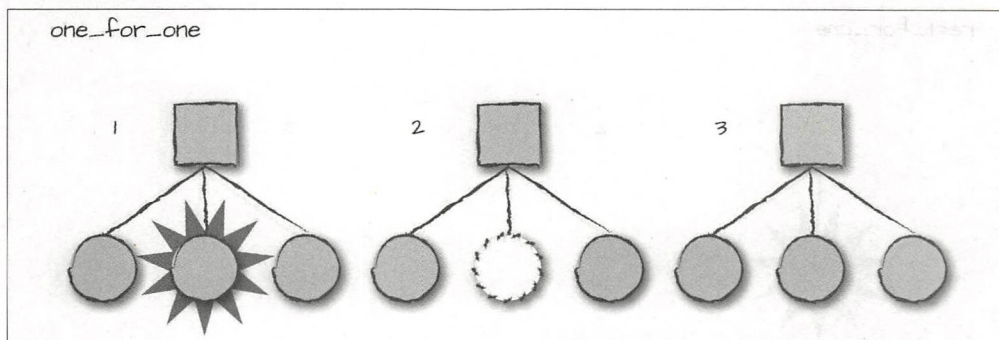


图 8-7: 一对一

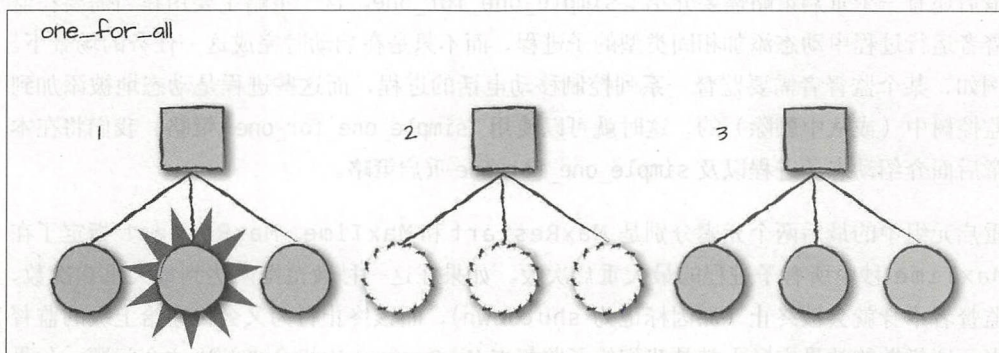


图 8-8: 一对全

使用 `rest_for_one` 策略时 (参见图 8-9), 当某个进程崩溃后, 当初在该进程之后启动的所有进程也会被终止并重启。如果你是按照依赖顺序启动进程的, 则可以使用此策略。我们在 `frequency_sup` 的示例中, 首先启动过载事件管理器, 然后才启动频率分配器。当频率分配器在的频率用光时, 会向过载事件管理器发送请求。所以, 如果过载管理器崩溃并正在重启时, 频率服务器可能会出现丢失发送的请求的风险。在这种情况下, 我们希望先终止频率分配器, 然后再依顺序重启过载管理器和频率分配器。

如果发送到频率分配器的报警丢失也没关系 (因为请求是异步的), 我们可以使用 `one_for_one` 策略。或者可以更进一步, 把调用过载管理器实现触发警报和清除警报的操作改为使用同步方式, 而不是异步。在这种情况下, 如果过载管理器崩溃并正在重启, 如果频率分配器尝试对其进行同步调用, 则也会被终止, 否则正常运行。频率分配器在没有把频率分配光的情况下不会触发或清除警报, 因而可能会继续正常运行。正如我们所看到的, 没有“一刀切”的解决办法, 一切都取决于你的需求以及你希望赋予系统怎样的行为。

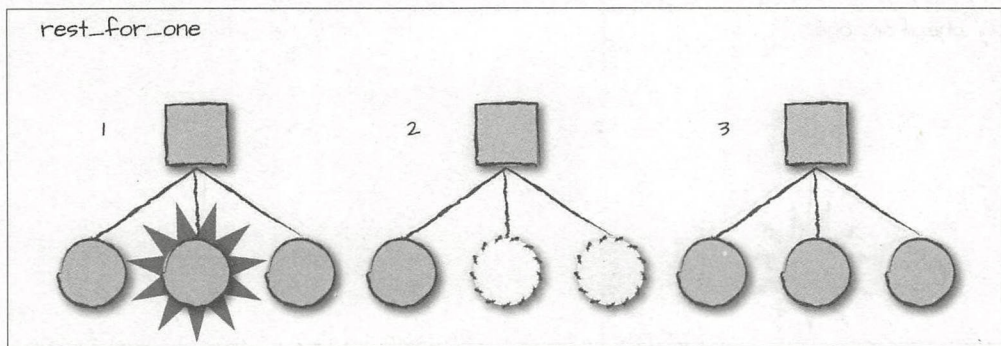


图 8-9: 余对一

最后还有一个重启策略需要介绍：`simple_one_for_one`。这一策略主要用在当需要在监督者运行过程中动态添加相同类型的子进程，而不只是在启动时完成这一任务的场景下。例如，某个监督者需要监督一系列控制移动电话的进程，而这些进程是动态地被添加到监控树中（或从中删除）的，这时就可以使用 `simple_one_for_one` 策略。我们将在本章后面介绍动态子进程以及 `simple_one_for_one` 重启策略。

重启元组中的最后两个元素分别是 `MaxRestart` 和 `MaxTime`。`MaxRestart` 指定了在 `MaxTime` 秒中所有子进程的最大重启次数。如果在这一秒数范围内达到最大重启次数，监督者本身就会被终止（原因标记为 `shutdown`），而该终止行为又会传递给上级的监督者。这样做的效果实际上就是我们给了监督者 `MaxRestart` 次机会来解决这个问题。如果在 `MaxTime` 秒内仍然发生崩溃，这意味着重启不能解决问题，因此监督者会将问题升级到其上级监督者，期待也许有可能解决此问题。

看图 8-2 所示的监督树。如果手机监督者（`phone supervisor`）下的手机 FSM 因频率处理器中的数据损坏而崩溃了怎么办？无论重启它们多少次，它们都将继续崩溃，因为问题在于频率分配器——一个位于监控子树之外的工作者进程。要解决这种性质的循环重启，办法是把故障提升到更高层次去处理。如果我们允许电话监督者终止，则顶级监督者会收到其退出信号，并在重启电话监督者之前先重启频率服务器和事件管理器这两个工作者进程。通过重启有希望清除损坏的数据，使手机 FSM 能够恢复正常功能。

你应当恰当设计启动顺序和与之相关的重启策略，这是用好监督者的关键。虽然你永远无法完全预测有哪些因素会导致你的进程异常终止，但是通过善用重启策略，可以从良性环境重建进程状态。与其简单地将状态持久化存储并假设它完好从而能够靠它从崩溃中恢复，更好的做法是从原始数据源中获取所需的数据进而重建状态。

例如，如果是由于临时性的传输错误导致了你的工作者数据错乱进而崩溃，那么重新读

取可能会解决问题。监督者将重启工作者，然后它成功地重传了正确的数据并继续运行。并且由于系统可能已经记录了崩溃信息，开发人员可以调查其原因，修改代码以适当地处理它，并准备和部署新版本，以确保未来类似的传输错误不会对系统产生负面影响。

但在其他情况下，恢复可能就没这么简单了。更复杂的传输错误可能会导致工作者反复崩溃，从而导致监督者重启工作者。但由于重启无法解决这一问题，客户端那里的监督者最终会达到重启次数阈值并终止自身。这反过来影响了顶级监督者，最终达到自己的重启次数阈值，并且通过终止自己将使整个虚拟机停下来。当虚拟机终止时，我们在第 11 章中将介绍的监视机制中的 heart 部分检测到节点已关闭，于是调用特定的 shell 脚本。此脚本中的恢复操作可能只是重启 Erlang VM 那样简单，也可能像重启计算机一样复杂。重启可能会通过重置反复出现传输错误的硬件链路而解决了问题。如果没有，在几次重启尝试后，脚本可能决定不再继续尝试重启，而是提醒操作员，请求人工干预。

但愿某个负载平衡器能跳出来将请求转发给冗余硬件，为最终用户提供无缝的服务。如果没有，你将会在午夜时分收到疲惫无援的一线支持工程师通知你系统中断的电话。在各种情况下，都应将崩溃记录下来，这样做有希望得到足够的数据支撑你调查并解决错误，换言之，请确保数据流入系统之前先经过必要的检查，从而从入口处避免如传输错误导致的损坏数据进入。我们会在第 13 章中学到分布式架构和容错。但现在，让我们专注于单个节点级别上的恢复。接下来进入的是子进程规格。

183

子进程规格

子进程规格包含监督者启动、停止和移除其子进程所需的全部信息。子进程规格是一个格式如下的元组：

```
{Name,StartFunction,RestartType,ShutdownTime,ProcessType,Modules}
```

或者，在 Erlang 18.0 或后续版本中，子进程规格还可以是一个映射，其类型定义为：

```
child_spec() = #{id => child_id(),           % mandatory
                  start => mfargs(),          % mandatory
                  restart => restart(),       % optional
                  shutdown => shutdown(),     % optional
                  type => worker(),           % optional
                  modules => modules()}       % optional
```

元组的元素有：

Name

任何有效的 Erlang 项，用于识别子进程。它在监督者中必须是独一无二的，但在同一节点内的不同监督者间可以重用。

StartFunction

一个格式为 {Module, Function, Args} 的元组，其直接或间接地调用了—个行为模式的 start_link 函数。监管者只能启动符合 OTP 规范的行为模式，而且它们有责任确保行为模式能够与监督者进程链接起来。你不能将普通 Erlang 进程链接到监督树，因为它们并不处理系统调用。

RestartType

告诉监督者如何对子进程的终止做出反应。将其设置为 permanent 可确保子进程总是重新启动，无论其终止是正常的还是非正常的。将其设置为 transient 则只有子进程是非正常终止时才重新启动。如果你不想在终止后重新启动子进程，请将 RestartType 设置为 temporary。

ShutdownTime

ShutdownTime 是一个正整数，表示以毫秒为单位的时间，或原子 infinity。它表示了从监督者发出 EXIT 信号到 terminate 回调函数返回之间允许的最长时间。如果子进程负载过重花费了超过此限度的时间，则监督者将无条件地终止子进程。请注意，仅当子进程捕获 exit 时才会调用 terminate。如果你是一个脾气暴躁的人，或者不需要行为模式自己做清理时，可以指定 brutal_kill，允许监督者使用 exit(ChildPid, kill) 无条件终止子进程。

选择 ShutdownTime 要小心，永远不要将 worker 设置为 infinity，因为这可能会导致 worker 挂起在其 terminate 回调函数中。想象一下，你的 worker 正试图与一个已停转的硬件进行通信，完成后你的系统才会重新启动。但你将永远不会得到回应，因为系统的该部分已关闭，于是这将致使系统无法重新启动。如果你必须这么做，那就使用一个很大的数字，最起码这最终将允许监督者终止 worker。另一方面，对于那些自身同样也是监督者的子进程，虽非强制但选择 infinity 很常见，这使得它们拥有足够长的时间关闭它们也许巨大的子树。

ProcessType 和 Modules

这些是在软件升级期间用来控制在升级过程中哪些进程被暂停以及如何被暂停的。ProcessType 可以是原子 worker 或 supervisor，而 Modules 是实现该行为模式的模块列表。在频率分配服务器例子中，我们将把 frequency 包含于其中，而对于咖啡机的例子，我们将把 coffee_fsm 包含于其中。如果你的行为模式包括特定于该行为模式的库模块，而你担心行为模块的升级与库模块之一不兼容，那么也请包括它们。例如，如果更改了 hw 接口模块中的 API 以及调用它的 coffee_fsm 行为模式，则必

须同时对两个模块进行原子升级，以确保 `coffee_fsm` 不会调用旧版本的 `hw`。通过列出这两个模块，你能够同时替换它们，从而避免问题。但是如果你没有列出 `hw`，就像在我们的示例中一样，那么必须确保任何升级都是向后兼容的，并妥善处理旧的和新的 API。我们在第 12 章中将更详细地介绍软件升级。

如果你在编译阶段无法确定你的模块怎么办？想想事件管理器，它的启动过程不涉及任何事件处理器。当你不知道执行软件升级时将运行什么，将 `Modules` 设置为原子 `dynamic`。当使用动态模块时，监督者将在需要的时候向行为模式模块发送请求获取其模块名称。

在查看接口和回调详细信息之前，先用学到的东西测试一下我们的例子。看看它们的子进程规格，我们看到，过载事件管理器和频率服务器是永久（`permanent`）工作进程，并给定了 2 秒用于执行 `terminate` 函数。我们启动监督者及其子进程，看到它们都已正确启动了。在 shell 命令 4 中，我们停止了频率服务器，但由于其 `RestartType` 设置的是 `permanent`（永久），因此监督者立刻就重启了它。我们通过检索新频率服务器进程的 `pid` 并注意到它与从 shell 命令 2 返回的原始服务器的 `pid` 不同，验证了 shell 命令 5 中的重新启动。在 shell 命令 6 中，我们显式地杀死了频率服务器，然后 shell 命令 7 表明，它再次重启了：

185

```
1> frequency_sup:start_link().
{ok,<0.35.0>}
2> whereis(frequency).
<0.38.0>
3> whereis(freq_overload).
<0.36.0>
4> frequency:stop().
ok
5> whereis(frequency).
<0.42.0>
6> exit(whereis(frequency), kill).
true
7> whereis(frequency).
<0.45.0>
8> supervisor:which_children(frequency_sup).
[{frequency,<0.45.0>,worker,[frequency]},
 {freq_overload,<0.36.0>,worker,[freq_overload]}]
9> supervisor:count_children(frequency_sup).
[{specs,2},{active,2},{supervisors,0},{workers,2}]
```

在 shell 命令 8 中，`which_children/1` 返回一个元组列表，包含 `ChildId` 对应的 `pid`，而 `worker` 或 `supervisor` 表示了其角色，以及 `Modules` 列表。如果你的监督者有很多子进

程使用此功能时要小心，因为它会消耗大量的内存。如果你从 shell 中调用此函数，请记住，结果将存储在 shell 历史记录中，不会被垃圾回收，直到历史记录被清除。

```
supervisor:which_children(SupRef) -> [{Id, Child, Type, Modules}]
supervisor:count_children(SupRef) -> [{specs, SpecCount},
                                       {active, ActiveProcessCount},
                                       {supervisors, ChildSupervisorCount},
                                       {workers, ChildWorkerCount}]
supervisor:check_childspecs(ChildSpecs) -> ok
                                       {error, Reason}
```

函数 `count_children/1` 返回一个属性列表，其中覆盖了监督者的子进程规格以及所管理的进程。其中的元素包括：

`specs`

子进程总数，既包含活跃（active）的也包含非活跃的。

`active`

活跃（active）运行中的子进程数。

186 **workers 和 supervisors**

相应类别的子进程的数。

最后，在开发和排查子进程规格以及启动方面的问题时，`check_childspecs/1` 函数非常有用。它能够验证一个子进程规格列表，如果有错误则返回 `error`，如果没有发现问题则返回原子 `ok`。

监督者规格很容易编写。也正因此，它们也很容易被写错。太多次，程序员选择了脱离当前应用程序运行实际情况的配置值，或者直接从其他应用程序中复制规格，更有甚者，使用的是各类编辑器模板框架里的默认值。在设计启动和重启策略时，必须认真设计正确的监督结构，并且必须支持容错和冗余。这些任务包括以依赖关系顺序启动进程，并设置重启阈值，这些阈值会将问题传播到层级更高的监督者，使得当较低层级的监督者遇到无法解决的问题时，能将问题提升到更高级别的监督者层面进行处理。

动态子进程

了解完 `init/1` 回调函数返回的监督者规格之后，你一定已经发现了，到目前为止，我们面对的子进程都是那种与监督者一起启动的静态子进程。但是还可以用另一种方案：当启动监督者时，在 `init/1` 调用中动态创建子进程规格列表。例如，我们可以检查活跃

移动设备的数量，并针对每一个活跃移动设备启动对应的工作者进程。我们已经处理了 worker 生命周期的结束阶段（通过让 worker 短期存在，即电话被关闭，则 worker 也相应被终止），但是在生命周期的开始阶段我们还没有实现类似的灵活性。如果启动监督者后有移动设备入网怎么办？这类问题的解决方案是使用动态子进程，如图 8-10 所示。

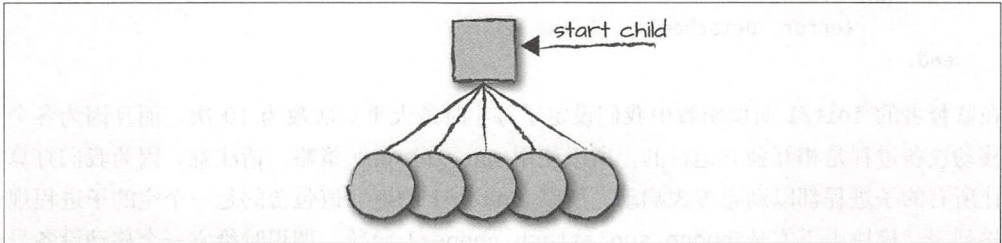


图 8-10: 动态子进程

让我们开始一个空的监督者，其唯一的责任是动态启动和监视控制移动设备的 FSM 进程。我们将使用的 FSM 是在第 6 章“电话控制器”一节介绍过并留作练习的那一个。如果你还没有解决该练习题，可以从本书的代码库中下载代码。该代码包括一个手机模拟器 *phone.erl*，它能启动指定数量的移动设备并让它们互相呼叫。我们会让电话监督者成为频率监督树中的一个子节点。下面来看看 *phone_sup* 模块的代码：

```
-module(phone_sup).
-behavior(supervisor).

-export([start_link/0, attach_phone/1, detach_phone/1]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init([]) ->
    {ok, {{one_for_one, 10, 3600}, []}}.

attach_phone(Ms) ->
    case hlr:lookup_id(Ms) of
        {ok, _Pid} ->
            {error, attached};
        _NotAttached ->
            ChildSpec = {Ms, {phone_fsm, start_link, [Ms]},
                          transient, 2000, worker, [phone_fsm]},
            supervisor:start_child(?MODULE, ChildSpec)
    end.
```

```

detach_phone(Ms) ->
    case hlr:lookup_id(Ms) of
        {ok, _Pid} ->
            supervisor:terminate_child(?MODULE, Ms),
            supervisor:delete_child(?MODULE, Ms);
        _NotAttached ->
            {error, detached}
    end.

```

在监督者的 `init/1` 回调函数中我们设定了每小时最大重启次数为 10 次，而且因为各个移动设备进程是相互独立运行的，所以使用 `one_for_one` 策略。请注意，因为我们打算让所有的子进程都以动态方式启动，所以 `init/1` 的返回值包含的是一个空的子进程规格列表。模块中下方是 `phone_sup:attach_phone/1` 函数，调用时给它一个移动设备号码 `Ms`，它会检查该号码是否已经在网络上注册过。如果没有，它创建一个子进程规格，并调用 `supervisor:start_child/2` 来启动它。

我们来试试这些代码。在 shell 命令 1 到 3 的交互过程中，我们启动了监督者并初始化了注册归属地数据库 `hlr`（在第 2 章的“ETS: Erlang 元素存储”小节我们曾介绍过）。在 shell 命令 4 和 5 中我们启动了两个手机进程，并提供了简单的数字作为电话号码参数。在 shell 命令 6 中，我们使用电话 2（被进程 `P2` 控制）向电话 1 发起呼叫。两个电话的 FSM 都开启了调试输出，使得我们可以追踪 `phone` 模块里的电话 FSM 和电话模拟器之间的交互过程。根据调试输出，可以看到电话 2 拨打了电话 1。电话 1 收到了入站呼叫并拒绝了，呼叫的终止使得两个电话都恢复到了空闲状态（由于模拟器是随机响应的，所以运行代码时可能会得到不同的结果）：

```

1> frequency_sup:start_link().
{ok,<0.35.0>}
2> phone_sup:start_link().
{ok,<0.40.0>}
3> hlr:new().
ok
4> {ok, P1} = phone_sup:attach_phone(1).
{ok,<0.43.0>}
5> {ok, P2} = phone_sup:attach_phone(2).
{ok,<0.45.0>}
6> phone_fsm:action({outbound,1}, P2).
*DBG* <0.45.0> got {'$gen_sync_all_state_event',
                  {<0.33.0>,#Ref<0.0.4.55>},
                  {outbound,1}} in state idle
<0.45.0> dialing 1
*DBG* <0.45.0> sent ok to <0.33.0>

```



```

and switched to state calling
*DBG* <0.43.0> got event {inbound,<0.45.0>} in state idle
*DBG* <0.43.0> switched to state receiving
ok
*DBG* <0.43.0> got event {action,reject} in state receiving
*DBG* <0.43.0> switched to state idle
*DBG* <0.45.0> got event {reject,<0.43.0>} in state calling
1 connecting to 2 failed:rejected
<0.45.0> cleared
*DBG* <0.45.0> switched to state idle
7> supervisor:which_children(phone_sup).
[{2,<0.45.0>,worker,[phone_fsm]],
 {1,<0.43.0>,worker,[phone_fsm]]}
8> supervisor:terminate_child(phone_sup, 2).
ok
9> supervisor:which_children(phone_sup).
[{2,undefined,worker,[phone_fsm]],
 {1,<0.43.0>,worker,[phone_fsm]]}
10> supervisor:restart_child(phone_sup, 2).
{ok,<0.53.0>}
11> supervisor:delete_child(phone_sup, 2).
{error,running}
12> supervisor:terminate_child(phone_sup, 2).
ok
13> supervisor:delete_child(phone_sup, 2).
ok
14> supervisor:which_children(phone_sup).
[{1,<0.43.0>,worker,[phone_fsm]]}

```

看看我们例子中的其他 shell 命令。你会看到很多用于从子进程规格列表启动、停止、重启和删除子进程的函数，其中一些在 `phone_sup` 模块中也使用到了。注意我们是如何调用 `supervisor:which_children/1` 得到工作者列表的。在 shell 命令 8 处我们终止了子进程，注意，在 shell 命令 9 的输出中可以看到该进程的子进程规格依然位于整个子进程规格列表中，并未删除，但是 pid 处值为 `undefined`。这意味着子进程规格仍然存在，但是进程没有运行。现在，在 shell 命令 10 中我们只需使用子进程的 Name 就可以重启它。

请记住，这些函数调用不使用 pid，而只能使用唯一的名称来标识子进程规格。这是因为子进程崩溃并重启后它们的 pid 可能会改变。然而，它们的唯一名称不会变。

一旦监督者已经存储了子进程规格，我们可以使用唯一名称来重启对应的子进程。要从子进程规格列表中删除它，需要首先终止子进程，如 shell 命令 12 所示，之后我们在 shell 命令 13 中调用 `supervisor:delete_child/2`。看看 shell 命令 14 中的子进程规格，可以看到手机 2 的规格已被删除。

简单一对一

当一个监督者下的所有进程共享相同子进程规格时，可以使用 `simple_one_for_one` 重启策略。我们的电话监督者示例符合此要求，因此我们使用此策略重写它。在这样做的同时，我们添加了 `detach_phone/1` 函数，稍后我们会进行解释。我们将把对 `hlr:new()` 的调用移到监督者的 `init` 函数中：

```
-module(simple_phone_sup).  
-behavior(supervisor).  
  
-export([start_link/0, attach_phone/1, detach_phone/1]).  
-export([init/1]).
```

```
start_link() ->  
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).
```

```
init([]) ->  
    hlr:new(),  
    {ok, {{simple_one_for_one, 10, 3600},  
         [{ms, {phone_fsm, start_link, []},  
          transient, 2000, worker, [phone_fsm]}]}}.
```

```
attach_phone(Ms) ->  
    case hlr:lookup_id(Ms) of  
        {ok, _Pid} ->  
            {error, attached};  
        _NotAttached ->  
            supervisor:start_child(?MODULE, [Ms])  
    end.
```

```
detach_phone(Ms) ->  
    case hlr:lookup_id(Ms) of  
        {ok, Pid} ->  
            supervisor:terminate_child(?MODULE, Pid);  
        _NotAttached ->  
            {error, detached}  
    end.
```

如果你详细查看了代码，大概会发现，`simple_one_for_one` 与我们之前为动态子进程使用的重启策略之间的差异。第一个变化是启动子进程时传递的参数不同。在监督者的 `init/1` 回调函数中，子进程规格 `{phone_fsm, start_link, ChildSpecArgs}` 没有指定参数（`ChildSpecArgs` 为 `[]`），而前面例子里 `phone_fsm:start_link(Args)` 函数取了一个参数，`Ms`。由于子进程是动态的，因此它们需要通过函数 `supervisor:start_child(SupRef, StartArgs)` 启动。此函数取其第二个参数——应该是一个列表，与子

进程规格中的参数列表相附加，调用 `apply(Module, Function, ChildSpecArgs ++ StartArgs)`。

对于电话 FSM，子进程规格中的 `ChildSpecArgs` 为空列表，所以将 `[Ms]` 作为第二个参数 (`StartArgs`) 传递给 `supervisor:start_child/2` 的结果是调用了 `phone_fsm:start_link(Ms)`。还值得注意的是，负责初始化 ETS 表的 `hlr:new()` 调用位于 `init/1` 中，目的是使当前监督者成为表的所有者。

第二个不同之处在于，在 `simple_one_for_one` 策略中，指代子进程不再是通过其名称，而是其 `pid`。如果你学习了 `detach_phone/1` 函数，会注意到这一点。你还会在代码中注意到，我们正在终止子进程，但并没有从子进程规格列表中删除它。我们不必这么做是因为当它被终止时它的子进程规格也会被自动删除。因此，函数 `supervisor:restart_child/1` 和 `supervisor:delete_child/1` 是不允许用的。只有 `supervisor:terminate_child/2` 可以用。对监督者进行的测试表明一切都在预料之中：

```
1> frequency_sup:start_link().
{ok,<0.35.0>}
3> simple_phone_sup:start_link().
{ok,<0.40.0>}
4> simple_phone_sup:attach_phone(1), simple_phone_sup:attach_phone(2).
{ok,<0.43.0>}
5> simple_phone_sup:attach_phone(3).
{ok,<0.45.0>}
6> simple_phone_sup:detach_phone(3).
ok
7> supervisor:which_children(simple_phone_sup).
[{undefined,<0.42.0>,worker,[phone_fsm]},
 {undefined,<0.43.0>,worker,[phone_fsm]}]
```

191

一旦我们分离 (`detach`) 了电话进程，它就不会再出现在监督者的子进程里了。这是 `simple_one_for_one` 策略特有的功能，使用除此之外的其他策略，你既需要终止子进程还必须删除子进程。另一个区别体现在关闭时，`simple_one_for_one` 监督者下运行着大量彼此独立的子进程很常见（常见于每个并发请求就创建一个子进程的情形），当关闭监督者时，终止子进程的过程是并发的而且顺序不确定。这是可以接受的，因为在这种情况下确定与否已经无所谓了，而且极有可能根本不需要这方面的保障。最后，`simple_one_for_one` 监督者能够满足有大规模动态子进程的场景，因为它使用 `dict` 键值字典库模块来存储子进程规格，而其他监督者类型使用的是普通的列表。虽然其他监督者在子进程数量不多时可能更快，但如果动态子进程的启动和终止频率较高，它们的性能会迅速恶化。

保持 ETS 表存活

你应该还记得，每个 ETS 表都会与创建它的进程相链接。如果该进程正常或异常终止，ETS 表将被删除。你可以在创建表时使用 `heir` 选项，或者在 `terminate` 函数中调用 `ets:give_away/3` 函数来转移所有权。然而，更简单的解决方案是——不要将 ETS 表放在其拥有者进程中，而是放在监督者进程中。选择那个监视当前用表进程的监督者进程作为 ETS 表的拥有者，这样一来如果监督者终止，则可以确保使用 ETS 表的进程也必然终止。此方法要求表具有公共（public）访问权限，以便非所有者进程也可以读取和写入。在我们的例子中，我们把负责映射 pid 到数字以及将数字映射到 pid 的 ETS 表放到了那里。如果监督者被终止或关闭，访问表的所有进程也会同样被终止。这种方法的主要缺点是如果 ETS 表中的数据损坏，则需要重新启动监督者才能清除它。如果你使用这种方法，请记住这一点。

要消化的内容还真是有点多。在继续下面的内容之前，我们来回顾一下用于管理动态子进程的功能 API。别忘了，`terminate_child/2`、`restart_child/2` 和 `delete_child/2` 不能与 `simple_one_for_one` 策略一起使用：

```
192 supervisor:start_child(Name, ChildSpecOrArgs) -> {ok, Pid}
                                     {ok, Pid, Info}
                                     {error, already_started |
                                     {already_present, Id} |
                                     Reason}

supervisor:terminate_child(Name, Id) -> ok
                                     {error, not_found | simple_one_for_one}

supervisor:restart_child(Name, Id) -> {ok, Pid}
                                     {ok, Pid, Info}
                                     {error, running | restarting |
                                     not_found | simple_one_for_one}

supervisor:delete_child(Name, Id) -> ok
                                     {error, running | restarting |
                                     not_found | simple_one_for_one |
                                     Reason}
```

整合全部

在封装此示例之前，让我们创建一个顶级的监督者 `bsc_sup`，它会启动 `frequency_sup` 和 `simple_phone_sup`。我们将使用 `phone.erl` 电话测试模拟器来测试系统，可以指定电话号码和每部电话应该尝试呼叫的次数，然后进行随机呼叫、接听呼叫或者拒绝呼叫。顶级监督者的代码如下：

```
-module(bsc_sup).
```



```

-export([start_link/0, init/1]).
-export([stop/0]).

start_link() ->
    supervisor:start_link({local,?MODULE}, ?MODULE, []).

stop() -> exit(whereis(?MODULE), shutdown).

init(_) ->
    ChildSpecList = [child(freq_overload, worker),
                     child(frequency, worker),
                     child(simple_phone_sup, supervisor)],
    {ok,{rest_for_one, 2, 3600}, ChildSpecList}.

child(Module, Type) ->
    {Module, {Module, start_link, []},
     permanent, 2000, Type, [Module]}.

```

我们选择 `rest_for_one` 策略，因为如果电话或电话监督者终止，我们不想影响频率分配器和过载处理程序。但是，如果频率分配器或过载处理程序终止，我们想重启所有的电话 FSM。每小时最多允许两次重启，如果超出此阈值则将问题升级，由 `bsc_sup` 的上级监督者（之类的，或者其他进程）去处理。

假设频率服务器中数据损坏导致电话 FSM 崩溃。当 `simple_phone_sup` 一个小时内终止三次后，由于超过其最大重启次数阈值，`bsc_sup` 将终止其所有子进程，包括频率服务器在内的进程都随它一起关闭了。重启可能有望解决此问题，使得电话恢复正常工作。在即将到来的章节中我们会展示应当如何处理这种故障升级。在此之前，我们使用 `phone.erl` 模拟器，通过启动 150 部电话，各自尝试拨打 500 次电话，来测试我们的监督结构和电话 FSM：

```

1> bsc_sup:start_link().
{ok,<0.35.0>}
2> phone:start_test(150, 500).
*DBG* <0.107.0> got {'$gen_sync_all_state_event',
                  {<0.33.0>,#Ref<0.0.4.37>},
                  {outbound,109}} in state idle
<0.107.0> dialing 109
...<snip>...
*DBG* <0.92.0> switched to state idle
*DBG* <0.53.0> switched to state idle
3> counters:get_counters(freq_overload).

```

```
{counters, [{event, {frequency_denied, <0.38.0>}}, 27],
  {{set_alarm, {no_frequency, <0.38.0>}}, 6},
  {{clear_alarm, no_frequency}, 6}]}
```

为了简单，我们只保留了一条调试输出内容，其余都省略了。运行测试后，我们查阅计数器看到在运行期间，曾经 6 次用完了频率，并且触发了（并最终解除了）警报。在这 6 次间隔中间，有 27 次呼叫发起失败。检查日志时，可以获取到这些呼叫发起失败时的时间戳。假如这其中蕴含一定的模式特征，我们可以使用这些信息来改进不同时段下频率的可用性。

在进入下一节之前，如果你运行的测试结果还显示在你的计算机上，并且 shell 还开着，那么请尝试使用 `exit(whereis(frequency), kill)` 杀死频率服务器 3 次。这会使顶级监督者达到其最大重启次数阈值进而终止。请注意，当 FSM 在其 `terminate` 函数中尝试断开（detach）自身时，由于 hlr ETS 表已经不存在，导致出现 `badarg` 错误。错误之所以发生在电话 FSM 的 `terminate` 函数中，是因为监督者在终止电话 FSM 之前先终止了 ETS 表。这些错误报告可能会掩盖另一些更重要的错误，因此在 `terminate` 函数中把可能失败的调用包在 `try-catch` 里总是一个好主意，并会默认返回原子 `ok`。

194

非 OTP 兼容进程

只有基于 OTP 行为模式（或者遵循行为准则的）并能够处理和响应 OTP 系统消息的进程才能链接到 OTP 监督树。然而，有时候，我们不想使用行为模式只想使用纯进程，原因可能是性能问题或者是遗留代码。要解决这一问题，一种方式是通过使用监督者桥接器（supervisor bridges）实现我们自己的行为模式，另一种方式是使用工作者分裂并链接常规 Erlang 进程。

监督者桥接器

在 20 世纪 90 年代中期，当时的爱立信启动了一些下一代电信基础设施重大项目，OTP 还在开发中。这些系统的第一个版本虽然符合大多数设计原则，但并不符合 OTP 标准，因为 OTP 尚不存在。当 OTP R1 发布时，我们花了大量的时间开会讨论是否应该将先前这些系统迁移到 OTP——开会花的时间甚至比实际执行此项任务所花的时间还多。那时就像这样，当没有进展时，`supervisor_bridge` 行为就派上用场了。

监督者桥接器是一种允许将非 OTP 兼容的进程连接到监督树的行为模式。对于它的上级监督者来说，它就像一个普通的监督者，但是它与子进程之间的交互是使用 `start` 和 `stop` 函数。在图 8-11 中，监控树的右侧由 OTP 行为模式组成，左侧则连接非 OTP 的进程。

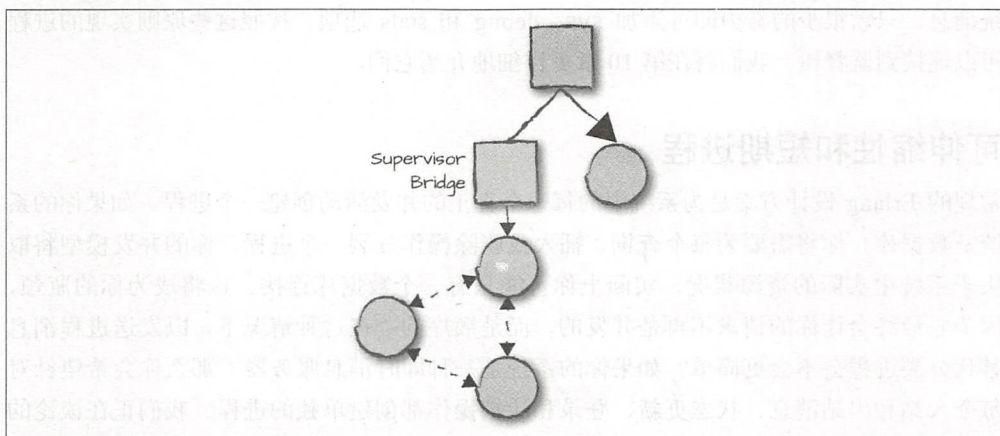


图 8-11: 监督者桥接器

通过调用 `supervisor_bridge:start_link/2,3` 并传入可选的 `NameScope`, 回调模块 `Mod` 以及参数 `Args` 可以启动监督者桥接器。这导致 `init(Args)` 回调函数被调用, 你可以在其中启动所有进程彼此链接的 Erlang 进程子树。如果 `init/1` 回调成功, 必须返回 `{ok, Pid, State}`。保存 `State`, 并将其作为第二个参数传递给 `terminate/2` 回调。

195

如果 `Pid` 进程终止, 则监督者桥接器将以相同的原因终止, 导致 `terminate/2` 回调函数被调用。在 `terminate/2` 中, 必须一个不漏地执行完所有必需的调用, 从而确保能够妥善关闭对应的非 OTP 兼容进程。此刻, 监督者桥接器的上级监督者负责接管和管理重启。如果监督者桥接器从其监督者接收到关闭消息, 则也会调用 `terminate/2`。尽管监督者桥接器本身通过 `sys` 模块完整地支持各种调试选项, 但是由它启动并链接的那些进程却并不支持代码升级和调试功能。监督功能受子树中所实现的功能制约。

```
supervisor_bridge:start_link(NameScope, Mod, Args) ->
    {ok, Pid} | ignore | {error, {already_started,Pid}}
```

```
Mod:init(Args)                -> {ok,Pid,State} | ignore | {error,Reason}
Mod:terminate(Reason, State) -> term()
```

添加非 OTP 兼容的进程

还记得吗, 监督者只接受兼容 OTP 的进程成为其监督树的一部分。它们包括工作者、监督者和监督者桥接器。但其实可以添加的还有最后一种: 像标准行为模式一样, 遵循 OTP 设计原则——但是实际上只遵循其中的一个子集——的一类进程。我们把这些遵循 OTP 设计原则, 但是不属于标准行为模式的叫作特殊进程 (special processes)。通过使用 `proc_lib` 模块你可以实现自己的特殊进程——能够启动进程并处理 `sys` 模块中的系

统消息。只需很少的努力即可添加 `sys`、`debug` 和 `stats` 选项。按照这些原则实现的进程可以连接到监督树。我们将在第 10 章更详细地介绍它们。

可伸缩性和短期进程

常规的 Erlang 设计方案是为系统中的每一个真正的并发活动创建一个进程。如果你的系统是数据库，你将需要为每个查询、插入或删除操作分裂一个进程。你的并发模型将取决于系统中实际的资源状况，实际上你可能只有一个数据库连接。这将成为你的瓶颈，因为它最终会让你的请求不再是并发的，而是顺序的。在这种情况下，以发送进程消息替代分裂进程会不会更简单？如果你的系统是一种即时消息服务器，那么你会希望针对每个人站和出站消息、状态更新、登录和注销操作都创建单独的进程。我们正在讨论的是同一时刻同一监督者下的几万甚至几十万个短期进程。在撰写本书时，对于拥有大规模动态子进程，并以非常短的间隔持续性频繁启动、停止的监督者，其伸缩性还不太好，因为在这种场景下，监督者内部出现了瓶颈。与其他使用列表存储子进程规格的监督者类型不同，`simple_one_for_one` 策略下的监督者伸缩性更好，因为它使用 `dict` 键值库模块存储子进程规格。但是尽管如此，它也还是有其局限性的。要想给出一条经验值用于界定允许的动态子进程启动和终止速率为多少是很难的，因为它受底层硬件条件、操作系统和内核以及进程本身的行为（包括考虑分裂过程中需要复制的数据）等诸多因素影响。一般而言这类问题很罕见，但是如果一个监督者消息队列增长到数千个消息时，那么你要知道你已受到影响。这个问题有两种解决方案。

简单的方法，如图 8-12 所示，是创建一个监督者池，使得每个监督者避免照顾过多的子进程。如果多个子进程需要与其他进程交互并且存活周期长，则推荐该方案。左侧的进程是调度员，管理协调监督者，当有必要时就创建新的。你可以使用最适合你需求的算法来选择池中的监督者，例如轮循（round robin）、一致性散列（consistent hashing）或随机。

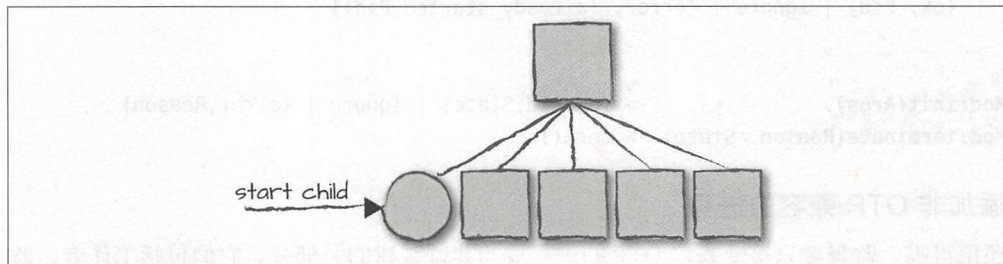


图 8-12：监督者池

许多人采用的第二种方法是让一个 `worker`（通常不是 `gen_server`）为每个请求 `spawn_link` 出一个非 OTP 型进程（参见图 8-13）。在消息服务器、Web 服务器和数据库中你

常会看到此策略。这种非 OTP 型进程通常执行一组顺序的同步式操作，一旦任务完成立即终止。这个解决方案为了速度和可伸缩性牺牲了 OTP 原则，但它确保你的进程与产生它的行为模式相链接；如果进程树关闭，链接的进程也将终止。

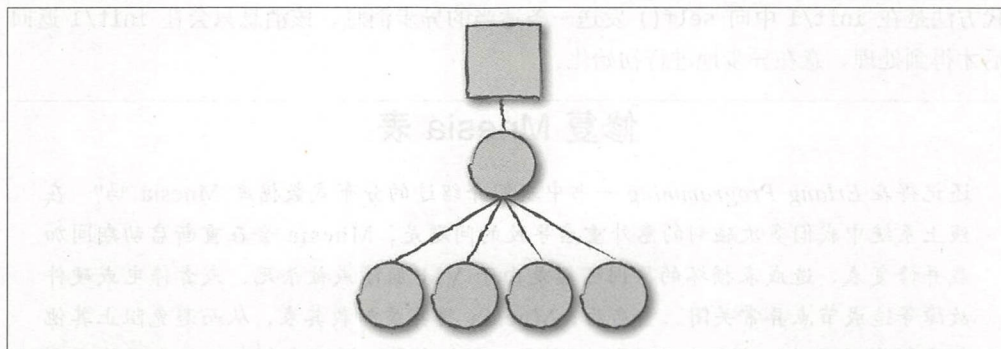


图 8-13: 链接到 worker

为什么要链接？不要忘记，你的系统将连续无重启地运行多年。你无法预测会碰到什么升级、新功能甚至异常终止。你最不想看到的就是由于上一次升级失败导致产生了一组无法控制的悬挂进程。你应当把不兼容 OTP 的子进程与父进程链接起来，这样一来如果父进程终止，子进程也会被终止。

197

多重监督策略

每个子进程可以与一个监督者或父进程相关联。OTP 监督树没有设计成支持一个行为模式同时属于两个进程组并且分别具有不同策略的情况。如果你遇到需要让多个进程监视同一个行为模式的场景，请使用链接与监视器机制，并确保只有一个行为模式负责处理重启策略。

确定性同步启动

还记得当你调用 `start` 或 `start_link` 启动行为时，进程的创建与 `init/1` 函数的执行是同步的吗？只有当 `init/1` 回调函数返回时，该函数才返回。这同样适用于监督者行为。任何行为在启动阶段的崩溃都将导致监督者失败，并终止其他所有已经启动的子进程。由于启动过程是同步的，因此如果希望降低启动和重启时的时间开销，请尽量减少 `init/1` 回调中的工作量。你需要确保该进程已重启并处于一致状态。如果启动涉及建立与远程节点或数据库的连接——而由于偶发性错误导致该连接可能会在之后出错——那么可以在 `init/1` 函数中开始连接，但不需要在其中等待连接完成。

198 > 一个用来推迟初始化的技巧是在 `init/1` 回调函数中将超时设置为 0。以这种方式设置超时会导致你的回调模块在 `init/1` 返回后立即收到超时消息，从而允许你异步地继续初始化行为模式。这一技巧可以用于各种情形，包括等待节点或数据库连接或任何其他非紧急的部分，这些任务都不需要你非得在 `init/1` 函数中完成。超时的一个更通用的替代方法是在 `init/1` 中向 `self()` 发送一条适当的异步消息，该消息只会在 `init/1` 返回后才得到处理，意在异步地进行初始化。

修复 Mnesia 表

还记得在 *Erlang Programming* 一书中我们介绍过的分布式数据库 Mnesia 吗？在线上系统中我们多次碰到的意外重启导致的问题是，Mnesia 会在重新启动期间加载并修复表，造成表损坏的原因可能是由于 VM 崩溃或被杀死，或者停电或硬件故障等造成节点异常关闭。重启后，Mnesia 将异步加载其表，从而避免阻止其他行为模式的启动。修复表需要很长时间，因为需要扫描日志和备份。如果你尝试从尚未完全加载的表中读取值，则调用将引发异常。

如果一个行为模式依赖于一组表，你可以通过在初始化行为模式时调用 `mnesia:wait_for_tables/2` 来应对这个问题。这种做法对于测试环境中的小表格是适用的，但在生产系统中，正在加载的数据可能很大。事实上，测试环境中的数据集通常是如此之小，以至于在没有调用 `wait_for_tables/2` 的情况下可能也不会有问题。但最坏的情况是，在大型线上系统重大中断后，你的监督者启动是否可以忍受几分钟时长的等待，原因是上次异常终止导致不得不修复 Mnesia 表？这会不会在其他地方造成不必要的邮件队列增长，甚至导致连锁效应？这些都是测试系统时必须验证的问题。

为什么同步启动很重要？想象一下，首先异步启动所有子进程，然后检查它们是否全部正确启动。倘若启动时出现的问题是由于进程的启动顺序或其各自的初始化回调中的表达式的执行顺序引起的，则要想重现导致启动错误的竞态条件可绝不是一件简单的事。另一个办法是启动一个进程，让它去初始化，并且只有在其 `init` 函数返回后才启动下一个进程。这将使你能够重现启动顺序错误导致的问题，而无须担心竞态条件。顺便说一句，这是我们使用 OTP 时的方式，其中将应用（将在第 9 章介绍）、监督者、同步式启动序列结合到一起，构成了一个“简单的核心”，能够保证你的系统的其余部分建立在一个坚实的基础上。

测试你的监督策略

在本章中，我们介绍了如何构建你的监督树，根据依赖关系分组和启动进程，并告诉了

你选择正确重启策略的方法。不要忽视或低估这些任务。虽然我们鼓励你消除防御性编程，并且鼓励你在碰到意外状况时把行为终止，但你需要确保已实现了错误隔离并能够从此异常中恢复。你可能会碰到遗漏依赖、选错重启策略，或者在错误的时间间隔内设置了过高（或过低）的允许重启次数等情况。你要如何测试这些场景并检测出这些设计缺陷呢？

非正常终止还是正常终止

本书作者之一曾经参与了一个项目，其中每个监督者管理的每个通用服务器都拥有一个 TCP 连接。当套接字由于连接错误而关闭时，会导致行为异常终止，进而重新启动，并尝试重新建立连接。每个网络连接错误虽然说是完全合规的，但这会增加计数器的异常终止次数，偶尔导致节点关闭。不巧的是，由于防火墙配置不当、路由器和负载均衡器故障导致网络连接有问题，或者甚至是系统管理员误操作导致网络电缆跳闸等，都会导致产生明显的网络连接问题。这些因素的存在除了引发网络中断之外，还可能进一步导致由此生成的大量错误报告掩盖了系统中发生的其他异常问题。考虑到这些状况可能会在正常操作下发生，所以结论是对于并非当前程序发起的套接字关闭也应该视为正常事件处理，不应当导致异常终止。

对于 Erlang 系统来说，所有正确编写的测试规范都会包含负面测试用例，其中必须通过模拟异常终止来验证恢复场景和监督策略。你需要确保系统不仅可以启动，还可以在发生意外事件时重启和自我修复。

在我们早期的第一套测试系统里，使用 `exit(Pid, Reason)` 来杀死特定进程并验证恢复场景。在以后的几年里，我们使用了 Chaos Monkey (<https://github.com/Netflix/SimianArmy>)，这是一款能够随机杀死进程的开源工具来模拟异常错误。对你的系统做压力测试的同时使用它能够更完整地模拟硬件和网络出故障时系统的表现。如果你的系统能活过这样的测试，那么它已经具备进入生产环境的品质了。



不要在公共场合谈论“杀死孩子”！

当我们还在为 OTP 的 R1 版本奋斗时，有一次我们中的一群人从办公室出发，乘坐通勤列车前往斯德哥尔摩。我们当时正在热火朝天地讨论如何轻松杀死“孩子”（即 Killing Children，实际上指的是杀死子进程），“孩子们”的死亡过程等，并且兴奋地说我们无须害怕，因为主管（即 supervisor，实际上指的是监督者）会捕捉逃跑者（trap exits，实际上指的是捕捉退出信号）并让它们从头再来（restart，实际上指的是重启）。我们对此非常兴奋地高声探讨，因为这在当时还是一种前卫的软件开发方法，而我们正处于学习和实践阶段。我们都在这次谈话中充满激情，竟然没有注意到坐在我们旁边的一些老太太脸上的惊恐

200

表情。直到我们最终下车时才注意到人群中蔓延的不安情绪，而伴随着我们的离开大家似乎都长舒了一口气。专业提醒：在公开场合，请注意措辞，不要说杀死（kill）它们，而是终止（terminate）。谨记这些让你广交朋友，而不是紧张地向法官解释自己的措辞，毕竟他们可能没有幽默感。

与传统方法相比又如何

与传统编程语言中的防御式编程方法相比，让监督者去处理错误是更好的选择吗？我们圈子里的传奇 Erlang 程序员的实践表明，这可以编写更少的代码，以让产品更快地上市。但我们引用的数据是来自直觉和非公开的研究。事实上，第一个研究来自爱立信，MD110 公司交换机中的大量功能被从 PLEX（一种当时使用的专有语言）改写为 Erlang。结果是代码量减少到原来的十分之一。由于担心没有人会相信这个结果，官方的立场是你可以实现相同的函数，将代码减少到原来的四分之一。之所以选择“四分之一”是因为它足够大，令人印象深刻，但也足够小，不会受到质疑。最终给我们准确答案的，是苏格兰赫里瓦特大学的一项研究——将 C++ 生产系统重写为 Erlang/OTP。其中一个系统是摩托罗拉的 Data Mobility（DM），这是一个处理紧急服务使用的双向无线电数字通信系统。DM 是已经以 C++ 实现了容错和可靠性的，它被以各种不同的方案在 Erlang 中进行了重写，目的是能够对各种版本进行比较和对比。

许多学术论文和讨论都是围绕这一研究而写的。其中一个有趣的发现是，其中一个 Erlang 实现方案使得代码减少了 85%。这部分解释了 27% 的 C++ 代码都是在进行错误处理和防御性编程，别的什么也没干。如果你将 OTP 作为语言库的一部分，那么需要编写的 Erlang 代码几乎低到只需 1%！

仅仅是使用监督者和 OTP 行为模式内置的容错性，与其他常规语言相比，就可以减少 26% 的代码。删除由内存管理组成的 C++ 代码的 11%，删除由高级通信组成的其他代码 23%——这些功能都包含在 Erlang 语义或 OTP 里了，再把声明式语义和模式匹配考虑在内，你很容易理解为什么减少 85% 的代码能成为可能。关于本研究，如果你想了解更多信息，可以阅读一两篇论文^{注1}，或者查看在线提供的演示文稿。

总结

前几章介绍了一些 OTP 工作者进程，在此基础上，本章介绍了如何将它们组合到监督树中。我们已经了解了依赖和恢复策略，以及它们如何允许你以通用的方式处理和隔离故障。因为有这套方法，你就不再需要试图在自己的代码中处理软件错误或错乱的数据了。

注 1 Nyström, J. H., Trinder, P. W., and King, D. J. (2008), “High-level distribution for the rapid production of robust telecoms software: Comparing C++ and ERLANG,” *Concurrency Computat.: Pract. Exper.*, 20: 941–968. doi: 10.1002/cpe.1223.

把注意力聚焦在乐观的情况，万一真的出现意外导致你的进程终止，让别人来处理这个问题。这个策略正是我们所说的故障安全（fail safe）。

表 8-2 中列出了监督者（supervisor）和监督者桥接（supervisor bridge）行为模式导出的函数及其相应的回调函数。在它们各自的手册页中你可以阅读到更多内容。

表 8-2: 监督者回调函数

监督者函数或动作	监督者回调函数
supervisor:start_link/2, supervisor: Module:init/1 start_link/3	
supervisor_bridge:start_link/2, Module:init/1, Module: supervisor_bridge:start_link/3	terminate/2

在继续阅读之前，你还应阅读本章中提供的示例代码，并查看网上的一些监督者实现的示例。这样做将有助于你了解如何在设计系统的同时将容错和恢复考虑进去。

接下来是什么

在第 9 章中，我们将介绍如何将监督树打包成一种称为 application（应用程序）的行为模式。application 包含了若干监督树，并提供了操作来启动和停止它们。它们被视为 Erlang 系统的基本构件。在第 11 章中，我们将介绍如何将若干 application（应用程序）打包成为一个发行版本（release），为我们提供一个 Erlang 节点。

OTP application

在前几章中，我们研究了一系列 worker 型的行为模式，并了解了如何将它们组合在一起形成监督树。在本章中，我们将探索 application（应用程序）行为模式，它使我们可以把监督树、模块以及其他种类的资源打包为一个半独立的单位，为大型 Erlang 系统提供基本构件。当你想把代码和配置文件等打包并分发给世界各地的人们使用时，OTP application 是一个不错的选择。

每个 Erlang 节点通常由一系列松散耦合的 OTP application 组成，这些 application 之间会进行交互。而 OTP application 的来源则有多种：

- 一些是随爱立信标准发行包而自带的，包括 *mnesia*、*sasl* 和 *os_mon*。
- 其他一些非爱立信官方的，但对许多 Erlang 系统又是必需的通用型 application 可通过商业渠道或开源渠道获得。此类通用型 application 包括用于警报的 *elarm*，用于指标收集的 *folsom* 和 *exometer*，以及用于日志的 *lager* 等。
- 一般而言，每个节点中还具有一个或多个非通用的 application，其中包含了系统的业务逻辑。这些通常是针对特定系统专门开发的，包含与业务相关的核心功能。
- 最后一种 OTP application 类别是那种用户向的 application，它们连同所依赖的部分一起，能够在 Erlang 节点上独立运行。将这类 application 捆绑打包后就形成了 release（发行包）。例子包括 Yaws Web 服务器、Riak 数据库、RabbitMQ 消息代理和 MongooseIM 聊天服务器等。顺便提一下，虽然并非是一般性做法，但有时可以通过将业务逻辑 application 和这些用户向 application 运行在同一个节点上来增强吞吐量和总体性能。

尽管来源多种多样，但所有 OTP application 都遵循相同的结构。我们在本章后续部分会讨论这一结构的细节。而自此，在本书的其余部分里，我们使用的术语“应用程序（application）”都是专门指 OTP application，而不是在更广义的层面上指代其他非 OTP 的一般性的 application。

OTP application 是如何运行的

可以把 application 视为一种将一系列资源打包制作成可重用组件的手段。资源不仅指模块、进程、注册名称、配置文件等，还可以包括其他非 Erlang 源代码及可执行代码，如 bash 脚本、图形、驱动程序等。虽然不同的 OTP application 包含不同的资源，执行不同的功能或服务，但对 Erlang 运行时系统而言它们看起来都一样；Erlang 运行时系统以相同的方式加载和运行它们，并允许 application 间相互访问和调用，以及终止。图 9-1 展示了各种组件是如何共同运行在 Erlang 运行时之上的。

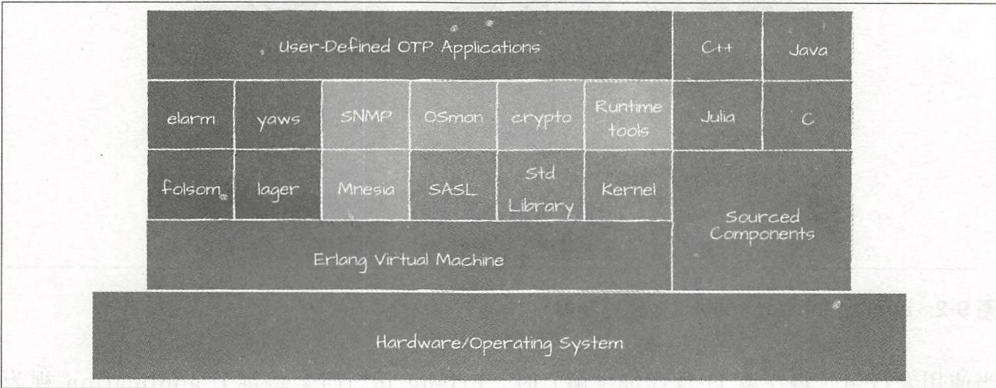


图 9-1: Erlang 分发包

application 可以被配置，并能够作为一个整体启动和停止。这使得系统能够轻松地管理许多监控树，使它们彼此独立地运行。一个 application 也可以依赖于另一个 application；例如，一个服务器端的基于 Web 的 application 可能依赖提供了 Web 服务器逻辑的 application，如 Yaws (<http://yaws.hyber.org/>) 之类。为了支持这样的 application 间的依赖关系，运行时必须能够以正确的顺序启动和停止各个 application。这实质上是一种清晰而优雅的封装能力，鼓励我们重用代码，其表达力在某种程度上已远远超出了普通模块。

application 可分为两个类别：*normal*（常规）application 和 *library*（库）application。常规 application 会启动一个顶级监督者，该监督者负责启动子进程，因而形成监督树。而库 application 则提供库模块给外部使用，但自身不启动监督者或进程；它们导出的函数可供运行于不同 application 中的 worker 或监督者进程调用。一个典型的库 application 的例子是 *stdlib*，其中包含了整个 OTP 标准库，如包括 *supervisor*、*gen_event*、*gen_server* 和 *gen_fsm* 等。

在 Erlang VM 内部会为每一个节点启动一个称为 *application controller*（应用控制器）的

进程。针对每个 OTP application，该控制器会启动一对名为 *application master*（应用主管）的进程。其负责启动和监视顶级监督者，并在顶级监督者终止时采取行动，示意图如图 9-2 所示。

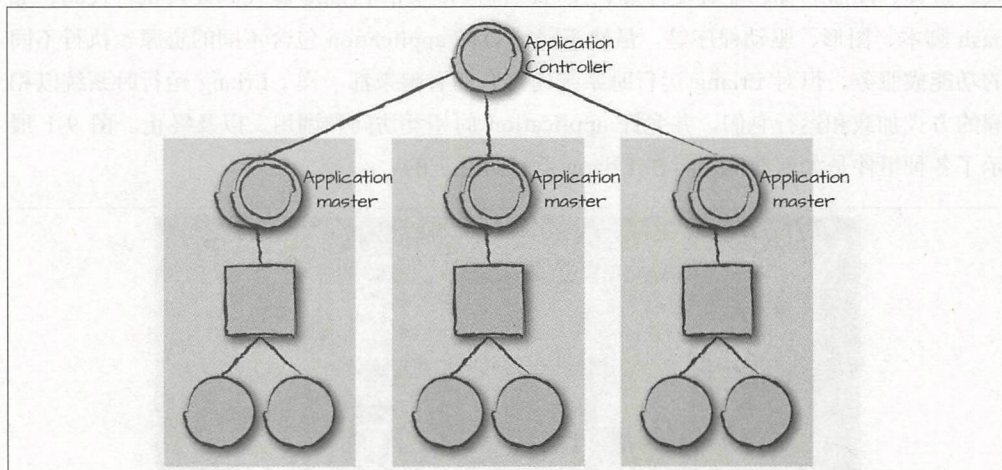


图 9-2: application controller（应用控制器）

当使用发行包（将在第 11 章详细讲解）时，Erlang 运行时会将每个 application 视为一个独立单元；每个 application 可以作为一个整体被加载、启动、停止和卸载。加载 application 时，运行时系统加载其所有相关模块并检查各个资源。如果任何一个模块丢失或损坏，启动将失败，并且该节点也被关掉。在启动 application 时，application master 进程创建顶级监督者进程，后者进而又启动了监督树的下级部分。如果监督树中有任何行为启动失败的话，整个节点也会被关闭。当停止 application 时，application master 进程会终止其所管理的顶级监督者，传递退出信号给监督树中所有的行为模式进程。最终，执行 application 卸载，运行时将会清除该 application 的所有已加载模块。

至此，我们对于各个部分如何衔接已经有一个较高层次的认识了，接下来让我们一起深入了解细节。

206 > OTP application 的结构

application 都会被放置在一个目录中，该目录遵循特别的结构和命名约定。各种周边工具都依赖于此结构，甚至包括发行包制作过程。一个典型的 application 的目录结构如图 9-3 所示，包含了 *ebin*、*src*、*priv* 和 *include* 等目录。

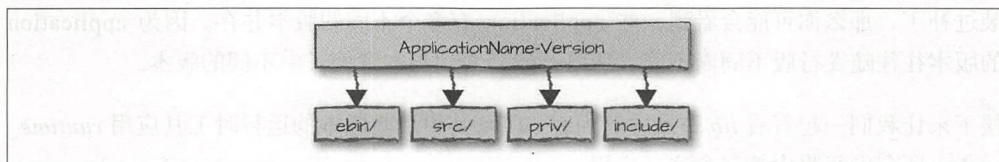


图 9-3: application 的目录结构

application 目录的名称由 application 名称后跟版本号构成。这使得你可以在同一个库目录中存放同一 application 的不同版本，并通过设置代码搜索路径指明正在使用哪个版本。application 的子目录通常包括如下几个。

ebin

包含 beam 文件和 application 配置文件——也被称为 app 文件。

src

包含 Erlang 源代码文件，以及一些你不希望其他 application 使用的头文件。

priv

包含 application 所需的非 Erlang 类文件，例如图像、驱动程序、脚本或专有配置文件。

include

包含导出的可由其他 application 使用的头文件。

其他非标准目录，如用于存储文档的 *doc* 目录，用于测试的 *test* 目录和用于示例的 *examples* 目录等，也可以作为 application 的一部分。而非标准目录与前面列出的标准目录的一个主要区别在于，访问标准目录时，运行时系统和工具允许你仅指明 application 名称即可，不需要提供版本号。例如，当你加载 application 时，代码搜索路径将直接指向当前选中的 application 版本的 *ebin* 目录。再比如，假设你想包括来自另一个 application 的 *.hrl* 文件，则 *makefile* 中的 *include* 路径将为你指向正确的版本。而对于非标准目录就没有这样的便利了，因此，你或者你使用的工具必须能够自行找到正确的路径。

让我们借助 OTP 分发中的一个实例来更深入地理解上述目录结构。请记住，Erlang 发行版里任何一个 OTP application 的目录结构都与你在自己系统中实现的 application 具有相同的结构。

◀ 207

转到 Erlang 根目录，然后从那里使用 *cd* 命令进入 *lib* 目录。如果你不确定 Erlang 安装到了哪个位置，启动 shell 并通过键入代码 *code:lib_dir()* 来确定 *lib* 目录的位置。*lib* 目录中包含了安装 Erlang 时所有的 application。如果你升级过先前安装的发行版或者安

355

装过补丁，那么你可能会看到一些 application 有多个不同的版本并存。因为 application 的版本往往随发行版不同而不同，所以可能会看到与本章例子中不同的版本。

接下来让我们一起来看看 *lib* 目录中的内容，以及其中最新版本的运行时工具应用 *runtime_tools*，所有发行版中都包含这一应用。

```
1> code:lib_dir().
"/usr/local/lib/erlang/lib"
2> halt().
$ cd /usr/local/lib/erlang/lib
$ ls
...<snip>...
appmon-2.1.14.2      erts-5.7.5      public_key-0.18
asn1-1.6.13         erts-5.8.1      public_key-0.5
asn1-1.6.14.1       et-1.4          public_key-0.8
asn1-2.0.1          et-1.4.1        reltool-0.5.3
common_test-1.4.7   et-1.4.4.3      reltool-0.5.4
common_test-1.5.1   eunit-2.1.5     reltool-0.6.3
common_test-1.7.1   eunit-2.2.4     runtime_tools-1.8.10
compiler-4.6.5       gs-1.5.11       runtime_tools-1.8.3
compiler-4.7.1       gs-1.5.13       runtime_tools-1.8.4.1
...<snip>...
$ cd runtime_tools-1.8.10/
$ ls
doc      examples  info      src
ebin     include  priv
```

doc 目录和 *info* 目录是非标准的，因此它们不属于 OTP 范畴（爱立信 OTP 团队用这两个目录来存储文档）。Erlang 开发人员常常增加一些特定于不同 application 的目录和文件，如 *test* 和 *examples* 目录。当发行版升级时，这些非标准目录和文件不一定会保留。如果你查看不同版本的 *runtime-tools* application，会发现早期版本中有一个 *info* 文件，但后续版本中则不复存在。

让我们把注意力集中于 OTP 的标准目录上。如果你 *cd* 进入 *runtime_tools* application 的 *ebin* 目录下并检查其内容，会看到一些 *.beam* 文件和一个 *.app* 文件，并且还可能有一个 *.appup* 文件。对于 *.beam* 文件，你可能已经知道其中存储的是 Erlang 字节码。而 *.app* 文件则是强制必有的应用资源文件，我们将在本章后面“application 资源文件”部分探讨其细节。*.appup* 文件则在你升级过你的某些 application 后才会出现。在第 12 章当我们详细介绍软件升级方面的主题时会涉及这个文件。

src 目录用于存放 Erlang 源代码。如果该目录中的某些模块需要用到一些不对外导出的 *.hrl* 文件，那么请把那些 *.hrl* 文件也放在这里。由于当前工作目录（CWD，Current

Working Directory) 会被自动囊括入 include 文件搜索路径中, 所以放在这里的文件在编译时能够被访问到。请注意, 你使用的构建系统需要在编译完成后把 *src* 目录中产生的 *beam* 文件移到 *ebin* 目录下, 这是它的职责。makefile 以及像 *rebar3* (第 11 章的 “rebar3” 一节将详细介绍) 之类的工具一般来说会为你做到这一点。

头文件中定义的宏和 record 通常属于接口描述的一部分, 其他 application 中的模块也需要能够访问到这些定义。*include* 目录的作用是在构建过程中使外部能够访问到存储在其中的 *.hrl* 文件。你可以使用下述指令, 在无须知道 *include* 目录路径或 application 版本的情况下直接访问 *.hrl* 文件:

```
-include_lib("Application/include/File.hrl").
```

其中 *Application* 部分是 application 的名称, 不包括版本号, 而 *File.hrl* 是头文件的名字。编译器会自动确定你正在使用的是哪个版本的 application, 从而找到其目录, 并自动读入该文件, 这样做的好处是你不需要随发行版而更改版本号。即使头文件不需要 *.hrl* 后缀, 但实践中最好还是包含它。发行包中的版本依赖处理方法会在第 12 章中介绍。

如果你在 Erlang 的 *lib* 目录下执行 `grep ^-include_lib ssl*/src/*.erl` 可检查所有安装在系统上的各个版本的 *ssl* application 的 *src* 目录。你会注意到一些模块中引用了来自其他 application (如 *public_key* 和 *kernel*) 的 *.hrl* 文件。还会有少数头文件是直接存储在 *src* 目录下的, 它们仅供 *ssl* application 自身使用。

priv 目录用于存放非 Erlang 型的资源。可以是链入式驱动程序、实现原生功能 (NIF) 的共享库、可执行文件、图形、HTML 页面、JavaScript 或 application 特定的配置文件, 基本上凡是 application 运行时需要, 但非 Erlang 相关的都存放在这里。以 *runtime_tools* application 为例, 其 *priv* 目录中包含其跟踪驱动程序的源代码和目标 (object) 代码。因为 *priv* 目录的路径随你运行的 application 版本不同而不同, 所以建议在你的代码中使用 `code:priv_dir(Application)` 来找到它。

通常部署在目标机器上时, 只需要 *ebin* 和 *priv* 目录就够了。这也许回答了你的问题——为什么强制要求把 application 资源文件放在 *ebin* 目录中而不是 *src* 目录中。看看其他标准发行包自带的 application, 你会注意到, 如果你确实没用到 *priv* 目录, 那么它也不是强制必备的。例如, *sasl* application 就没有 *priv* 目录, 像这样的 application 还有一些。

无论你是否打算把源代码和文档作为产品的一部分对外发布, 一般而言将它们全都打包进发行版并部署到目标机器不是一个好主意。因为在你升级 *beam* 文件时, 是不会自动检查源代码是否也随之更新的。有一次, 我们被要求解决一个中断故障, 我们阅读了生产机器上的很多代码后才意识到这些代码是第一版的代码, 因为系统经过一系列的修补、

重写、清理和重部署，这些遗留源代码早就已经过时了。当然，那些部署了新的 beam 文件的人很清楚目标机器上的源代码不是最新的这件事。而且他们也明白运维系统的人并不总是他们，但他们仍然假定我们这些做支持工作的人知道这一切并主动使用源代码仓库中的代码，或者假定我们会去询问。倘若你自己也是这样的人，那么智慧的忠告是，请永远从一开始就做这样的假设——假设你所编写和部署的系统是被一群杀人狂所维护和支持的，并且他们知道你的住址。午夜时分，由于你的代码中有 bug 导致服务中断，他们可不会和你讲道理，而是直接在黎明时分系统又挂了之后来敲你家的门……

并且，趁你现在注意力集中的时候，请记住，永远不要把编译器程序和你的系统源代码一同部署到生产系统中去。如果你这样做，那就是在自寻烦恼，因为你最终会因此尝试直接在目标机器上更改和编译代码来解决问题。好吧，假设代码的版本是对的（然而一般是错的），并且假设这样做确实能解决问题（然而一般解决不了），你最后还是难免会犯忘了把修改后的代码提交回真实的源代码库的错。不要忽略了这是在凌晨 3 点，你脑子里只想着回床上睡觉。切记，代码应该从代码库中取出，并且部署上线到系统之前必须先进行测试环境中进行测试。不论是多么紧急的修复，不要图省事，因为你的冒险将会让你付出代价，在另一天的白天，或者是晚上……

回调模块

application 行为模式与其他 OTP 行为模式没有区别。包含通用代码的 application 模块是 kernel 库的一部分，而所有的特定代码则要放在回调模块中（参见图 9-4）。

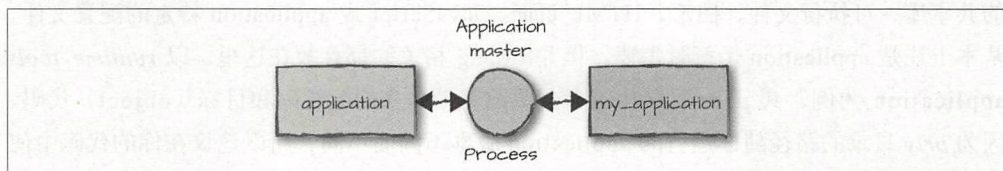


图 9-4: application 行为模式

210 > 在回调模块中必须包含 behavior 指令，并提供必需的和可选的各个回调函数。在所有 behavior 中，application 回调模块是最简单的。除非是需要实现分布式环境下的接管 (takeover) 和故障转移 (failover) 或是某类复杂的启动策略，否则一般而言你的 application 回调模块只需要写几行简单的代码就够了。

启动和停止 application

回调模块被调用的时机是在你的 application 启动时。要启动 application 需要使用

`application:start(Application)` 函数，其中 `Application` 指的是 `application` 的名称。此调用将加载所有打包在 `application` 内的模块，然后启动一些 `application` master 进程，其中一个进程会调用 `application` 回调模块中的 `Mod:start(StartType, StartArgs)` 回调函数。此 `start/2` 函数必须返回 `{ok, Pid}`，其中 `Pid` 是顶级监督者的进程标识符。如果 `application` 尚未加载，则启动 master 进程前会先调用 `application:load(Application)`。我们的 `application` 回调模块看起来类似这样：

```
-module(bsc).  
  
-behavior(application).  
  
%% Application callbacks  
-export([start/2, stop/1]).  
  
start(_StartType, _StartArgs) ->  
    bsc_sup:start_link().  
  
stop(_Data) ->  
    ok.
```

第一个参数 `_StartType` 经常被大多数 `application` 忽略；一般而言它是原子 `normal`，但如果我们运行的是具有自动故障转移（failover）和接管（takeover）功能的分布式 `application`，它的值也可能是 `{takeover, Node}` 或 `{failover, Node}`。我们在本章后面会讨论这些值。第二个参数 `_StartArgs`，来源于 `application` 资源文件中的 `mod` 键，我们会在本章后面的“`application` 资源文件”一节讨论。

图 9-5 中所示的 `application` 回调模块启动了顶级监督者。通常 `application` 回调模块中的 `start/2` 函数的功能仅仅只是调用由顶级监督者提供的 `start_link` 函数而已。例如，先前提到的 `bsc:start/2` 函数只是简单地调用了 `bsc_sup:start_link/0` 而已。

在我们的例子里，`bsc_sup:start_link/0` 返回 `{ok, Pid}`，而这也正是 `bsc:start/2` 返回的。另一个有效的返回值为 `{ok, Pid, Data}`，其中 `Data` 的内容被存储并且稍后传递给 `stop/1` 回调函数（参见图 9-6）。如果你没有返回任何 `Data`，则忽略传递给 `stop/1` 的参数即可（为了满足你的好奇心，实际上这种情况下参数会被绑定到 `[]`）。

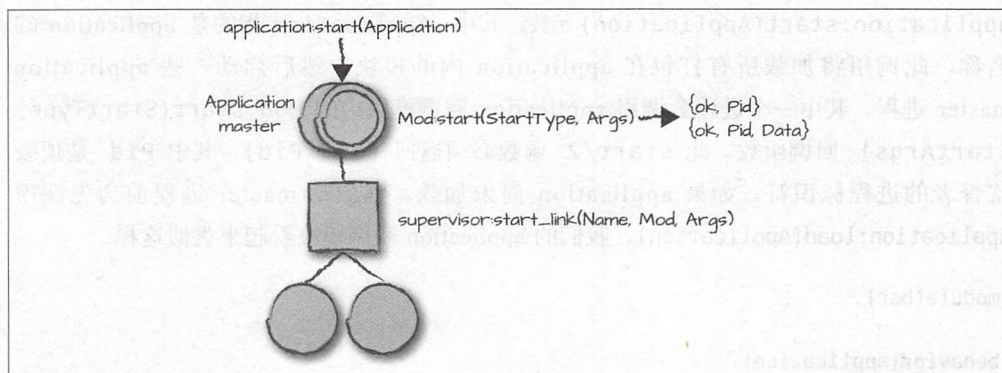


图 9-5: 启动 application

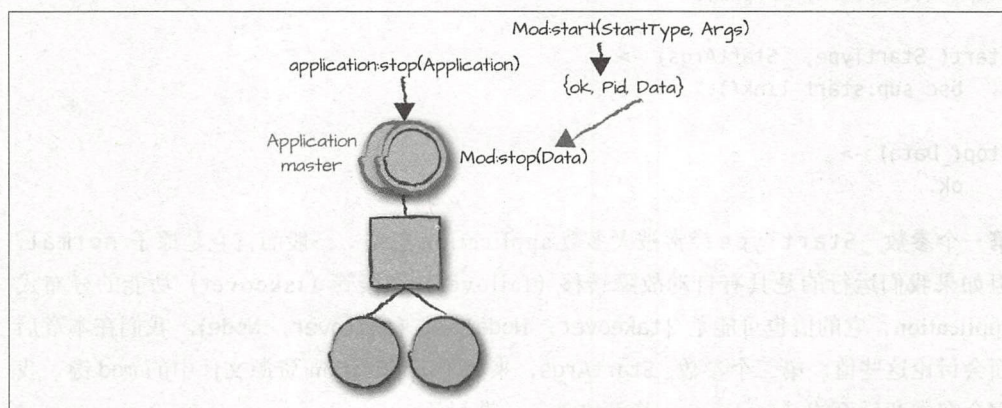


图 9-6: 停止 application

要停止 application, 请使用 `application:stop(Application)`。这样做会导致回调函数 `Mod:stop/1` 被调用, 时机是在监督树被终止——包括所有下级 worker 和监督者都被终止——之后。同样重要但属于可选的回调 `Mod:prep_stop/1` 也会在进程终止之前被调用。如果在终止你的监督树之前需要清理任何东西, `prep_stop/1` 是合适的地方。

我们来试试从标准 OTP 发行版加载、启动和停止 *sasl* application。根据你安装 Erlang 的方式不同, 启动 shell 时 *sasl* 可能会自动启动也可能不会。你可以通过输入 `application:which_applications()` 确定其是否启动了。在下面的示例里, 我们在 shell 命令 1 中执行了此操作后返回了一个元组列表。第一个元素是 application 的名称, 第二个元素是描述性字符串^{注 1}, 第三个元素则是一个表示 application 版本的字符串。当

注 1 CXC 实际上是爱立信内部的产品编号模式。据传, 凡是具有 CXC 编号的产品都会在瑞典境内森林中的秘密防核掩体里存储一份副本。

你启动 Erlang 时，其引导脚本（boot script）将决定启动哪些 application。如果你的 *sasl* application 位于此返回的元组列表中，在运行示例代码前请先停止它。就我安装的 Erlang 而言，它没有启动：

示例 9-1：载入一个 application

```
1> application:which_applications().
[{stdlib,"ERTS CXC 138 10","2.0"},
 {kernel,"ERTS CXC 138 10","3.0"}]

2> application:load(sasl).
ok

3> application:start(sasl).
ok

4>
=PROGRESS REPORT==== 17-Feb-2014::19:51:08 ===
    supervisor: {local,sasl_safe_sup}
      started: [{pid,<0.42.0>},
                 {name,alarm_handler},
                 {mfargs,{alarm_handler,start_link,[]}},
                 {restart_type,permanent},
                 {shutdown,2000},
                 {child_type,worker}]
...<snip>...

4> application:stop(sasl).

=INFO REPORT==== 17-Feb-2014::19:51:23 ===
    application: sasl
      exited: stopped
      type: temporary
ok
```

系统架构支持库（*sasl*）application 为你提供了用于构建、部署和升级 Erlang 发行包的工具集。它是最小的 OTP 发行包的一部分；连同 *kernel* 和 *stdlib* application，它们必须包含在所有符合 OTP 规范的发行包中。我们稍后会详细介绍这一切。

在我们的示例中，在 shell 命令 2 中加载了 *sasl*，并在 shell 命令 3 中启动它。你将注意到，当我们启动 application 时，shell 中会打印一长串进度（progress）报告（我们只保留了第一个输出，删掉了其余的）。*sasl* 启动其顶级监督者，后者进而又启动其他监督者和 worker。这些进度报告来自主要监督树中的监督者和 worker。我们在 shell 命令 4 中停止了 application。在继续向下阅读之前，请看看 *sasl* 回调模块中的源代码，位于文件 *sasl.erl* 中。如果你不确定在哪里能找到它，请使用 shell 命令 *m(sasl)*，它会告诉你 *beam* 文件所在的位置。从 *beam* 文件向上一级目录，然后再向下进入名为 *src* 的目录中

◀ 213

即可找到源代码。你需要在源代码中查看的函数是 `start/2` 和 `stop/1`。

application 资源文件

每个 application 都必须包含资源文件，又称为 *app* 文件。它包含由配置数据、资源和启动 application 所需的信息组成的规范。该规范是格式为 `{application, Application, Properties}` 的标记式元组，其中 `Application` 是表示 application 名称的原子，`Properties` 是一个由标记式元组组成的列表。

让我们先学习一下 *sasl* application 资源文件，然后再将它与我们手机的例子结合起来。这是 2.3.3 版本的 application；请注意，你的 app 文件的内容可能会根据你下载的版本不同而有所不同。一看它，你很可能立即注意到了 `mod`，其用途是指出 application 回调模块是哪一个以及应当传递什么参数给 `start/2` 回调函数：

```
{application, sasl,
  [{description, "SASL CXC 138 11"},
   {vsn, "2.3.3"},
   {modules, [sasl, alarm_handler, format_lib_sup, misc_sup, overload, rb,
              rb_format_sup, release_handler, release_handler_1, erlsrv,
              sasl_report, sasl_report_tty_h, sasl_report_file_h, si,
              si_sasl_sup, systools, systools_make, systools_rc,
              systools_relup, systools_lib]},
   {registered, [sasl_sup, alarm_handler, overload, release_handler]},
   {applications, [kernel, stdlib]},
   {env, [{sasl_error_logger, tty}, {errlog_type, all}]},
   {mod, {sasl, []}}}.
```

让我们自上而下浏览这些属性项。此属性项列表中包含的都属于标准属性项，但其实所有这些属性项都是可选的——如果列表中缺少了某一项，则会为该项设置一个默认值，但实际上除了极少数，几乎所有的 application 都自己设置好了各个属性项。标准属性项包括：

```
{description, Description}
```

其中 `Description` 是你自由设定的描述字符串。当你在 shell 中调用 `application:which_applications()` 时，你将看到它。默认值为空字符串。

214 {vsn, Vsn}

其中 `Vsn` 表示 application 版本的字符串。它应该反映目录的名称，并且应该由自动构建系统中的脚本设置，而不是手动设置。如果省略，默认值为空字符串。

`{modules, Modules}`

其中 `Modules` 是模块列表，默认为空列表。其用于指明创建发行版和加载 `application` 时使用的模块列表，在此列出的模块与 `ebin` 目录中包含的 `beam` 文件之间是一一对应的。如果你的模块的 `beam` 文件，位于 `ebin` 目录中，但未在此列出，则不会自动加载。^{注1} 此列表还用于检查各个 `application` 之间是否存在模块命名空间冲突，以确保名称的唯一性。

每个模块以一个对应模块名称的原子表示，如示例中的 `sasl`。Erlang 的 R15 及更早的版本中还允许指定模块版本 `{Module, Vsn}`，和出现在模块代码里的 `-vsn(Vsn)` 指令一样。但自那之后就不再是这样了。

`{registered, Names}`

其中 `Names` 是一个列表，包含了此 `application` 中运行的全部注册进程名称。包括此属性可以确保此 `application` 不会与其他 `application` 存在注册名称冲突问题。在这里少写一个名称并不会导致进程无法运行，但是当另一个 `application` 尝试注册同一个名称时，可能会触发运行时错误。如果省略此属性，其默认值为空列表。

`{applications, AppList}`

其中 `AppList` 指明了启动当前 `application` 前必须启动的所依赖的其他 `application` 的列表。所有 `application` 都依赖于 `kernel` 和 `stdlib` `application`，而且许多还依赖于 `sasl`。生成发行包时，此列表所表达的依赖关系决定了启动 `application` 的顺序。有时候，列表中只列出了一个 `sasl`，而其又是依赖于 `kernel` 和 `stdlib` 的。这么做行得通，但这使得系统更难维护和理解。如果省略此属性，其默认值为空列表，但这样做实在有些反常，因为这意味着当前 `application` 不依赖于其他任何 `application`。

`{env, EnvList}`

其中 `EnvList` 是 `{Key, Value}` 元组构成的列表，用于为 `application` 设置环境变量。而环境变量的值可以使用 `application` 模块的 `get_env(Key)` 或 `get_all_env()` 函数来获取，或者如果要读取的是其他 `application` 的环境变量而非当前 `application` 的，则可以使用 `get_env(Application, Key)` 和 `get_all_env(Application)`。环境变量也可以通过本章后面介绍的其他手段来设定。此属性默认为空列表。

`{mod, Start}`

其中 `Start` 是一个格式为 `{Module, Args}` 的元组，指明了 `application` 的回调模块以及传递给启动函数的参数。当 `application` 启动时，此元组的存在使得 `Module:start(normal, Args)` 被调用。省略此属性将导致 `application` 被视为库

注1 代码服务器可能在稍后你调用它时才加载。

application, 将由其他 application 的监督者或 worker 启动, 因而在启动时也不会为当前 application 创建监控树。

以下是一些其他的不包含在 *sasl.app* 文件示例中的属性, 但这些属性非常有用, 通常会出现其他 app 文件中:

`{id, Id}`

其中 *Id* 是表示产品标识符的字符串。只有极度沉迷配置管理的人才会使用该属性, 正如你看到的, OTP 团队不用它。默认值为空字符串。

`{included_applications, Apps}`

其中 *Apps* 是一个列表, 指明了当前 application 所包含的子 application。与其他 application 的区别在于, 子 application 的顶级监督者必须由其他监督者启动。本章稍后将介绍更多深入的应用。省略此属性将默认为空列表。

`{start_phases, Phases}`

其中 *Phases* 是一个元组列表, 由格式为 `{Phase, Args}` 的元组组成, 这其中的 *Phase* 是一个原子, 而 *Args* 可以是一个 Erlang 数据项。这一属性主要用于支持 application 的分阶段启动功能, 这项功能使得 application 能够与系统的其他部分相互同步并在后台启动 worker。在 `Module:start/2` 返回之前, 每一个阶段都会调用 `Module:start_phase(StartPhase, StartType, Args)`。其中 *StartType* 是原子 `normal` 或元组 `{takeover, Node}` 或 `{failover, Node}`。本章后面将详细介绍启动阶段 (Start Phases) 的概念。

基站控制器的 application 文件

了解了 app 文件的构造知识后, 我们也可以为我们的基站控制器 (bsc) 创建一个能用的 app 文件。除了 `description` 和 `application_vsn`, 我们还在 `modules` 属性中列出了组成 application 的所有模块。紧随其后, 我们还按照所注册的 worker 和监督者进程名称的列表设置了 `registered` 属性, 并在 `applications` 列表中声明 *bsc* application 依赖于 *sasl*、*kernel* 和 *stdlib*。不需要设置任何 `env` 环境变量, 但基于可读性原因, 仍然显式地将该列表保留为空。最后, application 回调模块 `mod` 被设置为 *bsc*, 并传入 `[]` 作为参数:

```
{application, bsc,
  [{description, "Base Station Controller"},
   {vsn, "1.0"},
   {modules, [bsc, bsc_sup, frequency, freq_overload,
               logger, simple_phone_sup, phone_fsm]},
   {registered, [bsc_sup, frequency, frequency_sup,
```



```

        overload, simple_phone_sup]],
    {applications, [kernel, stdlib, sasl]},
    {env, []},
    {mod, {bsc, []}}}.

```

既然 app 文件已完成，接下来就是将其放在 *ebin* 目录中，编译源代码，并确保生成的 *beam* 文件放在了 *ebin* 目录中。

启动 application

启动 Erlang 虚拟机时，请将你的 application 的 *ebin* 目录也加到路径中。这是测试时的好习惯，*bsc* 也许只是我们编写的众多 application 中的一个，考虑到这种情况下直接从它的 *ebin* 目录启动 Erlang 并非总是合适，所以我们需要考虑加载路径。但是之后制作发行包时加载路径问题会被自动处理，因而无须我们操心，但是此刻还享受不到这一点。在我们的示例中，我们添加了启动 Erlang 时使用的路径：

```
erl -pa bsc-1.0/ebin
```

你也可以使用 `code:add_patha/1` 在 Erlang shell 中添加路径。

让我们尝试启动 *bsc* application。在 shell 提示 1 中，我们失败是因为 *sasl*，它是 *bsc* 依赖的 application，但却尚未启动。为了避免这种情况，可以通过使用 `application:ensure_all_started/1` 来启动 application 所依赖的那些应用，然后再启动 application 本身，但为了简单起见，此处我们只需在 shell 命令 2 中启动 *sasl*，然后在 shell 命令 3 中再次启动 *bsc* 已足够解决这个问题。对于每个由顶级监督者 *bsc_sup* 所启动的子进程，我们都能从 *sasl* 处获得其进度报告。这是由于使用 OTP 行为模式而自带的功能：

```

1> application:start(bsc).
{error,{not_started,sasl}}
2> application:start(sasl).

```

...<snip>...

```
=PROGRESS REPORT==== 9-Jan-2016::18:47:09 ===
```

```

    application: sasl
    started_at: nonode@nohost

```

ok

```
3> application:start(bsc).
```

```
=PROGRESS REPORT==== 9-Jan-2016::18:47:40 ===
```

```

    supervisor: {local,bsc}
    started: [{pid,<0.51.0>},

```

```

        {id,freq_overload},
        {mfargs,{freq_overload,start_link,[]}},
        {restart_type,permanent},
        {shutdown,2000},
        {child_type,worker}]

=PROGRESS REPORT==== 9-Jan-2016::18:47:40 ===
    supervisor: {local,bsc}
        started: [{pid,<0.53.0>},
        {id,frequency},
        {mfargs,{frequency,start_link,[]}},
        {restart_type,permanent},
        {shutdown,2000},
        {child_type,worker}]

=PROGRESS REPORT==== 9-Jan-2016::18:47:40 ===
    supervisor: {local,bsc}
        started: [{pid,<0.54.0>},
        {id,simple_phone_sup},
        {mfargs,{simple_phone_sup,start_link,[]}},
        {restart_type,permanent},
        {shutdown,2000},
        {child_type,worker}]

=PROGRESS REPORT==== 9-Jan-2016::18:47:40 ===
    application: bsc
    started_at: nonode@nohost
ok
4> l(phone), phone:start_test(150, 500).
*DBG* <0.123.0> got { '$gen_sync_all_state_event' ,
        {<0.34.0>,#Ref<0.0.5.140>},
        {outbound,109}} in state idle
<0.123.0> dialing 109
...<snip>...

```

启动基站后，我们开始测试运行，启动了几百个随机相互拨打的手机。因为 phone 模块不是此 application 的一部分，所以在调用 phone:start_test/2 之前我们先手动加载了它。就我们的场景而言，即使不手动加载也没有影响，但是如果我们是在生产环境中以嵌入式模式运行，那么由于模块不会自动加载，将会导致出错。我们在第 11 章讨论发行包处理时，将进一步介绍各种启动模式。

如果你运行了此示例，请继续保持 Erlang shell 别关闭，然后键入 observer:start().，继续向下阅读。

observer 工具

observer 是一个图形工具，能够查看基于 Erlang 的系统的运行状态。它的功能更加完善，取代了你在旧版 Erlang 中可能用过的那些实用程序，包括进程管理器 *pman*、表可视化程序 *tv* 和 application 监视器 *appmon*。为了减少线上系统中的性能开销，你应该在单独的隐藏节点中启动 *observer* 工具，通过分布式 Erlang 方式连接到目标集群。因为我们的 *bsc* application 仍然处于开发阶段，所以我们可以偷偷懒，直接从本地启动 *observer*。

observer 窗口打开后将位于 System 选项卡处，你可以在其中查看常规信息，例如硬件结构、运行时系统版本以及与操作系统有关的数据。你还将看到与 CPU 和调度程序有关的详细信息、内存使用情况和常规的运行时统计信息。Load Charts 选项卡将实时绘制内存使用情况、调度程序利用率和 I/O 使用情况。虽然 *observer* 无法取代一些更深入的指标测量与监控系统，也无法存储历史数据，但它仍然有助于你了解正在开发的系统。

Applications 选项卡包含按字母序排列的 application 列表（参见图 9-7）。单击任何 application，你将看到相应的监督树，显示 worker 和监督者如何相互链接。我们来看 *bsc* application。你应该注意的第一件事是有两个 application master 进程。请注意，其中一个与 *bsc* 顶级监督者相链接，后者又与其他 worker 和监督者进程相链接。

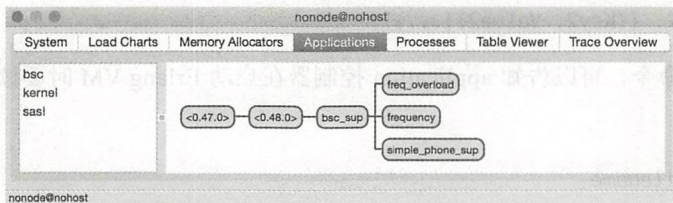


图 9-7: observer

随意单击一个进程，将弹出一个窗口，其中包含进程的基本信息、消息队列、字典和堆栈跟踪信息等。你也可以从 Processes 选项卡中查看同样的内容。Table Viewer 选项卡是表可视化器的移植版，允许你检查 Mnesia 和 ETS 表。最后，Trace Overview 选项卡是用于跟踪 BIF 和 *dbg* 的图形界面。你可以在“Observer 用户指南”和“参考手册”中阅读有关所有这些选项的更多信息。

环境变量

Erlang 在初始化 application 行为模式时，主要使用环境变量来获取配置参数。你可以设置、检查和更改这些环境变量。启动 Erlang shell，确保 sasl application 正在运行，然后键入 `application:get_all_env(sasl)`。此刻先不用担心这些环境变量的含义，稍后当我们介绍 sasl 报告时将介绍它们，但请注意它们与操作系统 shell 所支持的环境变量是不同的。现在，我们把注意力集中在如何设置和检索它们上。

如果你按照我们的建议调用了 `get_all_env(sasl)`，那么会看到它返回属于 sasl application 的环境变量。如果你想要一个特定的变量，如 `errlog_type`，那就使用 `application:get_env(sasl, errlog_type)`。如果检索环境变量的进程是 application 监督树的一部分，则可以省略 application 名称，只需调用 `application:get_all_env()` 或 `application:get_env(Key)`。

借助类似 `application:get_application()` 调用的功能，OTP 基于 Erlang 进程组组长（group leader）确定当前进程属于哪个 application。在我们的例子中，我们使用的是 shell，它不是 sasl application 监督树的一部分，所以必须指定 application。

这些环境变量是在哪里设置的呢？如果你查看 `sasl.app` 文件，将在 application 资源文件的 `env` 属性中发现它们。app 文件通常包含一些基本的值，但你可能需要根据系统需求和 application 的使用情况进行部分更改。解决这一状况的最好方式是使用系统配置文件完成。它是一个纯文本文件，带有 `.config` 后缀，包含格式如下的 Erlang 数据项：

```
{[{Application1, [{Key1, Value1}, {Key2, Value2}, ...]},  
 {Application2, [{Key2, Value2}|...]}].
```

通过使用以下命令，可以告知 application 控制器在启动 Erlang VM 时要读取哪个配置文件：

```
erl -config filename
```

其中 filename 是系统配置文件的名称，有或没有 `.config` 后缀都可以。

如果是在做原型、做测试或排除故障，你可以使用以下命令在命令行提示符中覆盖启动时在 application 和配置文件中设置的值：

```
erl -application key value
```

虽然这很方便，但这种方法不应该用于在生产系统中设定值。为了清楚起见，应坚持使用 app 文件和配置文件，因为对于调试或维护系统的人而言它们会是第一个关注点。

掌握了这些知识后，让我们自己编写 `bsc.config` 文件，其中包含频率分配器示例的频率，

并覆盖一些 *sasl* 环境变量：

```
[{sasl, [{errlog_type, error}, {sasl_error_logger, tty}]},  
 {bsc, [{frequencies, [1,2,3,4,5,6]}]}].
```

该文件覆盖了 *app* 文件中设置的 *errlog_type* 和 *sasl_error_logger* 环境变量。为了能从 *shell* 里测试配置参数情况，请启动 *Erlang* 节点并提供配置文件的名称，并将其放置在启动 *Erlang* 的同一目录中。在生产系统中，配置文件放在特定的发行目录中。我们在第 11 章中会更进一步进行介绍。

在以下命令中，启动 *erl shell*，我们进一步进行配置并覆盖 *sasl_error_logger*，将其值设置为 *false*。这样做可以关闭例子中余下操作触发的进度报告：

```
$ erl -config bsc.config -sasl sasl_error_logger false -pa bsc-1.0/ebin  
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> application:start(sasl).
```

```
ok
```

```
2> application:get_all_env(sasl).
```

```
[{included_applications,[]},  
 {errlog_type,error},  
 {sasl_error_logger,false}]
```

```
3> application:start(bsc).
```

```
ok
```

```
4> application:get_env(bsc, frequencies).
```

```
{ok,[1,2,3,4,5,6]}
```

```
5> application:set_env(bsc, frequencies, [1,2,3,4,5,6,7,8,9]).
```

```
ok
```

```
6> application:get_env(bsc, frequencies).
```

```
{ok,[1,2,3,4,5,6,7,8,9]}
```

在 *shell* 命令 1 中，我们启动 *sasl*，然后在 *shell* 命令 2 中查询其所有的环境变量。注意环境变量的最终值：

- *errlog_type* 是设置在 *bsc.config* 文件里的，覆盖了 *app* 文件中设置的值。
- *included_applications* 来自 *app* 文件。但它原本不是一个环境变量，是被 *application* 控制器转换而得的。
- *sasl_error_logger* 是在 *app* 文件中设置的，但在配置文件中被覆盖，并在启动 *Erlang* 时在 *UNIX* 命令行再次被覆盖。

频率服务器调用 *get_frequencies()* 检索频率时可以使用 *frequencies* 环境变量。请注意，我们不必在代码中指定 *application* 的名称，因为运行时可以根据发起调用的进程的

221 组长确定其所属的 application。在 frequency 模块的早期版本中，get_frequencies/0 函数内嵌了硬编码的频率列表。在这个例子中，经过我们的改写，无论使用或不使用 bsc.config 文件都能运行：

```
get_frequencies() ->
  case application:get_env(frequencies) of
    {ok, FreqList} -> FreqList;
    undefined      -> [10,11,12,13,14,15]
  end.
```

在交互示例的 shell 命令 5 处，我们在 Erlang shell 中直接设置了环境变量，然后在 shell 命令 6 中进行检索。其中 application 名称是可选的；如果没有提供，则设置和检索的环境变量将是属于发起调用的进程所属的 application。在我们的示例中，我们提供了 application 名，因为 shell 进程不属于 bsc application。



虽然你确实可以通过在 shell 中使用 application:set_env 函数修改环境变量，但建议你只针对自己编写的或者是非常熟悉的 application 才执行此操作。对于第三方 application，包括那些 Erlang 分发包中的 application，在它们已经开始运行后再去改变环境变量是危险的。因为你不知道这些 application 什么时候会读取这些环境变量，因此贸然更改可能导致其进入不一致的状态，并且出现难以预料的后果。而且你的修改在重启后也不一定能保留。做这个操作你要想清楚风险，记住只有当你完全了解 application 是如何读取和使用这些环境变量时才考虑这么做。

application 的类型与终止策略

当我们在示例 9-1 中停止 sasl application 时，收到了以下信息报告：

```
=INFO REPORT==== 17-Feb-2014::19:51:23 ===
  application: sasl
  exited: stopped
  type: temporary
```

你有没有发现其中 application 的类型 (type) 是 temporary？此类型决定了当你的 application 终止时，虚拟机和其他 application 的反应。使用 application:start(Name) 启动 application 时默认分配的类型是 temporary。但实际上存在以下三种可能的类型值。

temporary

当这种类型的 application 终止时，无论终止原因是什么，都不影响虚拟机和其他正在运行的 application。

222 transient

此类型的 application 以 normal 原因终止时，其他 application 不受影响。但以其他

异常原因终止时，将导致其他 application 也都终止，并且虚拟机也随之终止。此类型一般只有在编写自己的监督者行为模式（见第 10 章）时才会用到，因为监督者会使用 shutdown 作为终止原因。

permanent

只要 permanent 类型的 application 终止，无论原因是什么（正常或异常），所有其他正在运行的 application 也将与虚拟机一起终止。

创建自己的发行包时，会涉及这些选项，因为可以在引导脚本中设置它们。通常，在一个良好的 OTP 发行包中，所有 application 都倾向于是 permanent 类型的。application 中的顶级监督者永远不应当终止。如果真的出现这样的情形，则系统认为你的自动重启策略已经失效，所以整个节点都被关闭。但是，使用 application:stop/1 停止一个 application 时，无论其类型如何，都不会影响到其他 application。

分布式 application

OTP 具备强大的分布式机制，能够便捷地跨节点迁移 application。当你需要在集群中运行 application 实例的时候，一般而言它能够满足你的需要，并且当无法完全满足时，它还可以作为一个临时性方案先运行，等到更复杂的解决方案就绪后再替换。请注意，OTP 的分布式机制假定你的网络在大多数情况下都是可靠的，如果这一点不符合你的场景就得小心，确保在网络出现分区时你自己设计好了妥善的处理方案。

所有的分布式 application 都由称为 *distributed application controller* 的进程管理的，其在 dist_ac 模块中实现，并以相同的名称注册。在任何一个分布式节点上，如果你查看 kernel 监督树都会发现此进程的一个实例。

要想运行你的分布式 application，需要在 kernel application 中配置几个环境变量，以确保请求能被透明地转发到真正运行该 application 的节点，然后你需要反复测试。你必须指定运行该 application 时选择节点的优先顺序。一旦运行该 application 的节点出现故障，则该 application 将被故障转移（failover）到优先级列表中的下一个节点。如果集群中出现了具有较高优先级的新启动或新连接的节点，则 application 将迁移到该节点，这在 OTP 中被称为接管（takeover）。

假设我们的系统是一个由四个节点构成的集群组成的：`n1@localhost`、`n2@localhost`、`n3@localhost` 和 `n4@localhost`。让我们创建一个配置文件 `dist.config`，在其中设置 kernel 环境变量，使得我们的 `bsc` application 能够在该集群上分布式运行：

```
{[kernel, [{distributed, [{bsc, 1000, [n1@localhost, n2@localhost, n3@localhost]},
```

223


```

n4@localhost]]]],
{sync_nodes_mandatory, [n1@localhost]},
{sync_nodes_optional, [n2@localhost,n3@localhost,n4@localhost]},
{sync_nodes_timeout, 15000}}],
{bsc, [{frequencies, [1,2,3,4,5,6]}]}.

```

请注意，如果你打算运行此分布式的 *bsc* 示例，则可能需要替换 *dist.config* 文件中所有出现的“localhost”字符串为你的计算机名。

在 *kernel application* 的环境变量中，我们首先需要设置 *distributed* 项。该项中列举出了我们想分布式运行的 *application*，每一个 *application* 对应一个元组，其中包括 *application* 名称、超时值以及分布式节点或节点元组列表，其表明了我们运行 *application* 时希望以何种优先顺序选择节点。所以，这个列表：

```
[{bsc, 1000, [n1@localhost,{n2@localhost,n3@localhost},n4@localhost]}]
```

对应的 *application* 是 *bsc*，等待节点恢复的时间为 1000（以毫秒为单位），并且指定的节点优先级为：

```
[n1@localhost,{n2@localhost,n3@localhost},n4@localhost]
```

此优先级指定了 *application* 将从 *n1* 启动。如果该节点出现故障或关闭，分布式 *application* 控制器将等待 1 秒，然后将 *application* 故障转移至 *n2* 或 *n3*。这两个节点被置于同一个元组中，因而被赋予了相同的优先级。如果 *n2* 和 *n3* 都出现故障，控制器将检查 *n1* 是否已经恢复，如果仍然关闭，则 *application* 将故障转移到 *n4*。如果其他任一节点重新恢复，*application* 将在之后通过接管移动到具有最高优先级的节点上。

sync_nodes_mandatory（强制同步节点）和 *sync_nodes_optional*（可选同步节点）两个环境变量指定了要连接到的分布式系统节点。当系统启动时，分布式 *application* 控制器会尝试连接到它们指定的节点，并等待 *{sync_nodes_timeout, Timeout}* 环境变量中指定的毫秒数。如果在 *kernel* 环境变量中定义节点时省略了超时设定，则默认超时值为 0。

{sync_nodes_mandatory, NodeList} 环境变量定义了分布式 *application* 控制器必须同步的节点；当所有这些节点在 *Timeout* 毫秒内启动并相互连接时，系统才会启动。

环境变量 *{sync_nodes_optional, NodeList}* 指定了在系统启动时也可以试着连接的节点，但与强制节点不同，这些节点中任何一个超过了 *Timeout* 指定的时间未能成功加入指定的集群也不会导致系统无法启动。

224 > 弄懂环境变量设置的最佳方式是尝试设置一下 *dist.config* 配置文件。我们首先启动节点 *n2*：


```
$ erl -sname n2@localhost -config dist -pa bsc-1.0/ebin
```

此节点 (n2) 将等待 n1 节点 15 秒 (来自 `sync_nodes_timeout` 设置的值)。如果在该时间范围内无法连接到 n1, 则此节点终止, 并输出一大段难懂的错误消息。由于节点 n3 和 n4 也列入了可选同步节点, 所以假如 n1 顺利在超时时间内出现, 则 n2 也将同一时间段内等待 n3 和 n4 节点, 之后无论能否与 n3 和 n4 连接, n2 都将正常启动。

让我们再试一次, 但这一次, 在启动 n2 之前, 先启动 n1 和 n3:

```
$ erl -sname n1@localhost -config dist -pa bsc-1.0/ebin
$ erl -sname n3@localhost -config dist -pa bsc-1.0/ebin
```

这些节点将等待可选节点启动, 最长 15 秒。如果未成功, 这些节点也继续启动。你可以尝试从配置文件中删除 n4 (或不删除而是启动它), 避免超时等待。

当所有节点都启动后, 让我们在所有节点上启动 *sasl* 和 *bsc* application, 先是在 n3 上启动, 然后才是 n2 和 n1。请在所有三个 Erlang shell 中键入以下内容, 并在 shell 命令返回时注意:

```
application:start(sasl), application:start(bsc).
```

你会注意到, 执行此命令时, 如果是在 n2 和 n3 的 shell 中执行的, 则会卡住, 只有等到 n1 中的 *bsc* application 启动后才返回, 因为 n1 节点拥有运行 application 的最高优先级。如果启动 observer 并检查不同节点上的 Application 选项卡, 你将注意到只有 n1 上启动了监督树。看看 n2 和 n3 的进度报告, 你会注意到 *bsc* application 也启动了, 但却不带监督树。

保留节点 n2 和 n3, 现在使用 `halt()` shell 命令关闭节点 n1。

application 控制器将为 n1 重新启动等待 1000 毫秒。如果没有, 你将在 n2 或 n3 上看到 *bsc* application 启动的进度报告。在我们的配置文件中, 由于 n2 和 n3 具有相同的优先级, 所以到底会选择哪一个是不确定的。在图 9-8 中, 我们假设选择的节点是 n2。

n1 已经关闭了, 现在让我们关闭 n2 (或者如果 *bsc* application 是在 n3 上启动的, 则为 n3)。你将看到 application 故障转移到剩余的节点 (参见图 9-9)。使用 observer 检查监督树是否正确启动 (参见图 9-7)。重新启动刚刚关闭的节点, 观察发生的情况。你会注意到它卡住 15 秒, 等待 n1 重启。因为 n1 属于强制同步节点, 并且因为它没有重新启动, 所以此节点也无法重新启动。

◀ 225

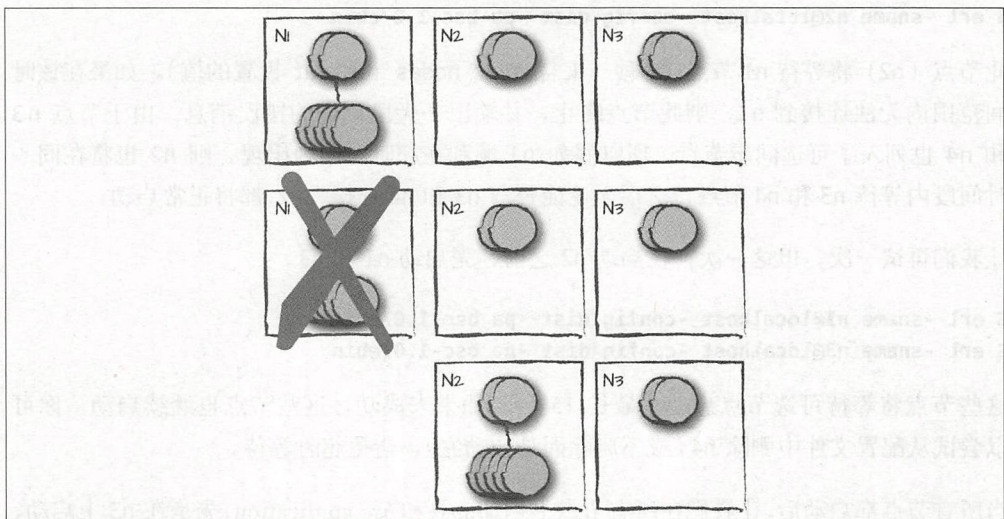


图 9-8: 不同优先级下的故障转移

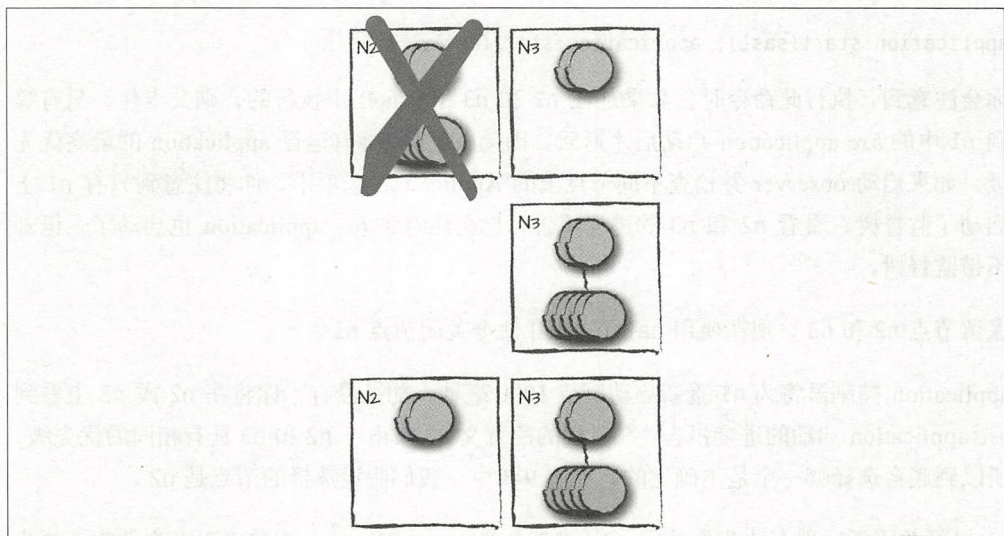


图 9-9: 相同优先级下的故障转移

226 在 15 秒内相继重新启动 n1 和 n2 (或者是 n3, 如果关闭的是 n3 的话)。两个节点都将等待 15 秒, 期待可选同步节点 n4 启动。在超时后, 使用 `application:start/1` 启动 `sasl` 和 `bsc`。正如你第一次启动集群时一样, `application` 将 (因为节点间需要相互协调而) 挂起直到 `bsc` 在 n1 上启动。当你在 n1 上启动 `bsc` 时, 将导致 n3 上的 `bsc` 被接管过来, 其行为模式被终止, 监督树被取消 (参见图 9-10)。

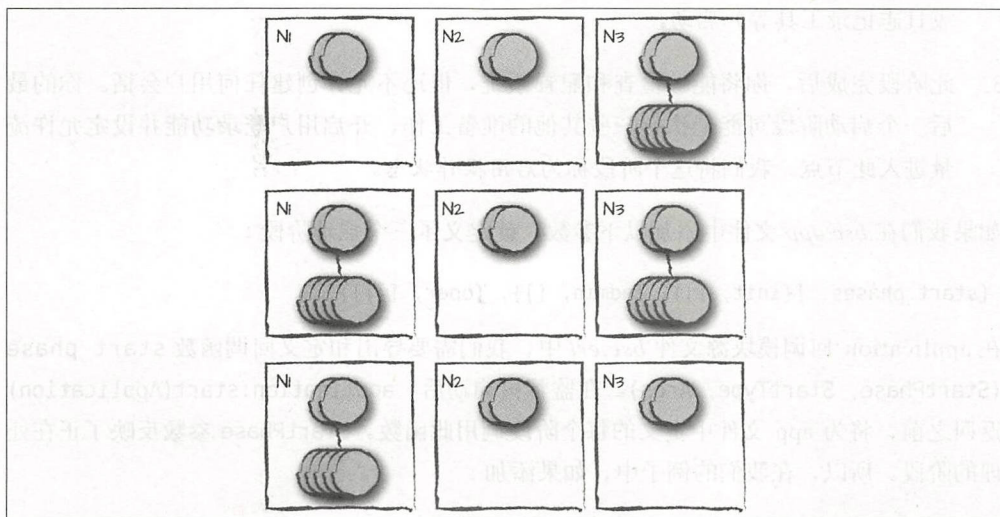


图 9-10: 接管 application

这种机制存在局限性，能够满足一些场景但对另一些场景则不适用。如果你选择这种方案，那么就意味着你得小心地确定强制同步节点。在设计无单点故障的系统时，你不应该假定或依赖某个节点在某个时刻的存活。如果系统依赖于某些服务，那么为了故障切换和接管机制能够正常运转，请通过 application 的分阶段启动机制，或是通过 worker 进程进行一些检查。虽然这一功能层可以很薄，仅需几百行代码，但是它是与 application 相关的。确保你已经深思过你的设计。在第 13 章讨论集群时，我们将分析其他分布式架构方案。

分阶段启动

一些系统非常复杂，要想在这种系统中简单地逐个启动 application 有困难。对于这样的系统，可以把多个 application 的启动过程划分为若干相关的阶段，并按阶段做同步。想象一下，有这样一个节点，它是处理即时消息的集群的一部分：

1. 在第一阶段，作为后台任务，你可能需要启动时加载包含路由和配置数据的所有 Mnesia 表。这可能需要一些时间，因为其中一部分表可能因为上一次突发的关闭或节点崩溃导致必须进行修复。
2. 一旦表加载好，接下来你的系统将转入下一阶段，即接受配置请求的状态——我们把这一阶段称为启动管理状态。在这一阶段，可能需要检查与同一联盟内其他集群的连接是否正常、对硬件进行配置以及等待系统的其余部分（例如身份验证服务器

227

或日志记录工具等) 启动。

3. 此阶段完成后, 你将能够检查和配置系统, 但还不允许创建任何用户会话。你的最后一个启动阶段可能是执行一些其他的准备工作, 开启用户登录功能并设定允许流量进入此节点。我们将这个阶段称为启用操作状态。

如果我们在 *bsc.app* 文件中添加以下参数, 就定义了三个启动阶段:

```
{start_phases, [{init, []}, {admin, []}, {oper, []}]}
```

在 *application* 回调模块源文件 *bsc.erl* 中, 我们需要导出和定义回调函数 *start_phase* (*StartPhase*, *StartType*, *Args*)。在监督树启动后、*application:start(Application)* 返回之前, 将为 *app* 文件中定义的每个阶段调用此函数。*StartPhase* 参数反映了正在处理的阶段。所以, 在我们的例子中, 如果添加:

```
start_phase(StartPhase, StartType, Args) ->  
    io:format("bsc:start_phase(~p,~p,~p).~n", [StartPhase, StartType, Args]).
```

到 *application* 回调模块 *bsc.erl* 文件中, 并使用修改后的 *bsc.app* 文件运行它, 我们将在启动 *application* 时得到以下事件序列。这两个文件都在代码库的 *start_phases* 目录中:

```
$ erl -pz bsc-1.0/ebin/ -pa start_phases/ -sasl sasl_error_logger false  
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)  
1> application:start(sasl), application:start(bsc).  
bsc:start_phase(init,normal,[]).  
bsc:start_phase(admin,normal,[]).  
bsc:start_phase(oper,normal,[]).  
ok
```

这里, *StartType* 的参数总是原子 *normal*, 表明这是一个正常的启动。各个阶段会发起同步或异步调用, 触发某些设计好的操作, 并设置内部状态, 从而允许或禁止节点处理请求。

228 > 当关闭系统时, 我们可以设定为停止执行新请求, 但允许所有现有请求执行到完成状态。这样一来, 系统将拒绝新的用户登录请求, 但却允许现有会话继续存在直到过期。等到所有用户都注销了或系统中所有余下的会话都超时后, 就可以关闭节点了。下一节我们将介绍一个使用了分阶段启动技术的简单示例。

内含型 application

在你的 app 资源文件中，你可以指定参数 `included_applications`。内含型 (included) application 的目录结构也放在 `lib` 目录中，和该发行包中的所有其他 application 一样。当主 application 启动时，所有内含型 application 都已加载但未启动。因为启动内含型 application 监督树的责任是交给主 application 的顶级监督者来完成的。你既可以把内含型 application 作为动态子进程启动，也可以将其作为静态子进程启动，不同的方式取决于在监督者 `init/1` 回调函数中返回怎样的子进程规格。

当启动内含型 application 时，你只需调用 application 模块中的 `start/2` 函数，并期待它返回 `{ok, Pid}` (而不是返回 `{ok, Pid, Data}`，因为我们无法在停止时将该 Data 传递给回调模块的 `prep_stop/1` 回调函数)，或直接调用顶级监督者的 `start_link` 功能。没有别的了，就这么简单！

在每个节点上，每一个内含型 application 只可以被其他 application 内含 (include) 一次。此限制避免了 application 命名空间中的冲突，确保每个模块和注册的进程 (本地或全局) 都是唯一的。如果你需要在同一个节点中启动多个相同的监督树，请将代码放在独立的库 application 中。不要把此 application 内含在依赖关系以外的其他位置，并确保 application 与本地和全局注册的进程没有名称冲突。

你可能会问自己，我们明明可以使用平坦的 application 结构单独启动各个 application，为什么还要这么麻烦使用内含型 application？答案与启动阶段有关。

内含型 application 的分阶段启动

你可以通过划分启动阶段使得启动过程中能与内含型 application 同步。由于内含型 application 的监督树是由主 application 启动的，为了能够调用到 application 回调模块中的 `start_phase/3` 回调函数你需要遵循几个步骤。

首先，在你内含的 application 的 app 文件中，确保已包括 `mod` 和 `start_phases` 参数。 ◀ 229
因为回调模块中会导出 `start_phase/3`，这里指定了 `mod` 才能找得到。注意在这个文件中指定的参数会被忽略，因为会使用 `start_phase/3` 中动态传入的参数。

最后，在你的顶级 application 中，除了定义启动阶段之外，还需要将 `mod` 参数更改为：

```
{mod, {application_starter,[Mod,Args]}}
```

将应用回调模块 `Mod` 和 `Args` 作为参数传入。OTP `application_starter` 模块提供了启动你的顶级 application 的逻辑，并且负责处理内含型 application 间的启动阶段协调。

这一过程很直接。顶级 application 的监督树启动各个内含型 application。顶级 application 的回调模块中的 `start_phase/3` 函数首先被调用，紧接着所有内含型 application 按照它们定义的顺序被遍历。如果一个或多个内含型 application 里定义了与顶级 application 中相同的启动阶段，则这几个内含型 application 的每一个的 `start_phase/3` 都会被调用。

顶级 application 的下一个启动阶段依照此规则递归式地触发。在内含型 application 中定义了却未在顶级 application 中定义的启动阶段永远不会被触发。

对所描述的内容的最好展现方式是借助例子。我们创建一个顶级 application `top_app`，其中内含了 `bsc` application。 `top_app` 回调模块负责启动所内含的 `bsc` application 的监督树：

```
-module(top_app).
-behavior(application).
-export([start/2, start_phase/3, stop/1]).

start(_Type, _Args) ->
    {ok, _Pid} = bsc_sup:start_link().

start_phase(StartPhase, StartType, Args) ->
    io:format("top_app:start_phase(~p,~p,~p).~n", [StartPhase, StartType, Args]).

stop(_Data) ->
    ok.
```

在顶级 application 的 `top_app.app` 文件中，我们定义了 `start`、`admin` 和 `stop` 阶段。它们与我们在本章前面“分阶段启动”一节介绍过的 `bsc` 启动阶段不同，之前的例子里是设置为 `init`、`admin` 和 `oper` 的。还请注意 `included_applications` 和我们为 `mod` 属性设置的值：

```
{application, top_app,
  [{description, "Included Application Example"},
   {vsn, "1.0"},
   {modules, [top_app]},
   {applications, [kernel, stdlib, sasl]},
   {included_applications, [bsc]},
   {start_phases, [{start, []}, {admin, []}, {stop, []}]},
   {mod, {application_starter, [top_app, []]}}
 ]
}.
```

启动阶段的工作内容如下。顶级 application 启动后，进而启动 `bsc` 监督树。一旦成功，`top_app` 中定义的第一个启动阶段 `start` 被触发。如果任何内含型 application，根据它们出现在 `included_applications` 列表中的顺序，也有这个阶段，则也被调用。如果

你在计算机上实验此过程，别忘了编译 *top_app* 目录的内容，并记得使用本章代码库中 *start_phases* 目录里的 *bsc.app* 文件：

```
$ erl -pz bsc-1.0/ebin/ -pa start_phases/ \
      -pa top_app/ -sasl sasl_error_logger false
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> application:start(sasl), application:start(top_app).
```

```
top_app:start_phase(start,normal,[]).
```

```
top_app:start_phase(admin,normal,[]).
```

```
bsc:start_phase(admin,normal,[]).
```

```
top_app:start_phase(stop,normal,[]).
```

```
ok
```

我们让这个例子足够简单，在不陷入业务逻辑的情况下演示了原理。在我们的示例中，调用了顶级 *application* 中的所有启动阶段，但对于内含型 *application* 只调用了 *admin* 阶段，因为这是它们之间唯一的共同定义了的阶段。

将监督者与 *application* 组合到一起

一些监督者回调模块只包含几行代码。如果你的 *application* 不需要处理复杂的初始化过程、启动阶段和分布式，而仅仅只需要启动顶级监督者，那么监督者回调模块中的代码确实会相当紧凑。一种常见的做法是将 *application* 和监督者的回调模块合二为一，因为它们的回调函数名称不重叠。虽然有些人会强烈反对这种做法，但是当阅读其他人的代码甚至是标准的爱立信发行包的代码时，你一定会遇到这种情况。

例如，*cd* 进入你安装的 OTP 的 *sasl* 目录，并看看 *sasl.erl* 文件。在撰写本书时，*sasl* *application* 的 2.6.1 版本中将其 *application* 模块中的监督者的 *init/1* 回调功能与 *application* 的 *start/2* 和 *stop/1* 回调函数组合到了一起。在这个例子中，开发人员只包含了 *-behavior(application).* 指令，但是实际上把 *-behavior(supervisor).* 指令包括进来也是可以的。唯一的副作用是编译器会警告你在同一个回调模块中使用了两个 *behavior* 指令。我们建议包括这两个指令，因为它有助于理解回调模块的用途。下面是一个简单的例子，将我们 *bsc* 中的 *supervisor* 和 *application* 回调模块组合到一起后看起来就像这样：

```
-module(bsc).
```

```
-behavior(application).
```

```
-behavior(supervisor).
```

```
-export([start/2, start_phase/3, stop/1, init/1]).
```

231

```

start(_Type, _Args) ->
    {ok, Pid} = supervisor:start_link({local, ?MODULE}, ?MODULE, []).

start_phase(Phase, Type, Args) ->
    io:format("bsc:start_phase(~p,~p,~p).",[Phase, Type, Args]).

stop(_Data) ->
    ok.

%% Supervisor callbacks

init(_) ->
    ChildSpecList = [child(freq_overload),
                     child(frequency),
                     child(simple_phone_sup)],
    {ok,{{rest_for_one, 2, 3600}, ChildSpecList}}.

child(Module) ->
    {Module, {Module, start_link, []},
     permanent, 2000, worker, [Module]}.

```

SASL application

在本章中，我们一直在告诉你查看 SASL 回调模块、app 文件、目录结构和监督树，但是我们还没有告诉你 SASL 实际做了些什么。

SASL 的含义是系统架构支持库。SASL application (*sasl*) 中包含了设计大规模软件时需要用到的一系列工具。它是最小 OTP 发行包中必须附带的 application（另外两个是 *kernel* 和 *stdlib*）。这是强制性规定，因为 *sasl* 中包含了用于分发制作和软件升级的所有常用库模块。

我们将在第 11 章中介绍发行包，在第 12 章中介绍软件升级。然而，SASL 并非仅仅止步于处理发行包和软件升级方面。在第 7 章的“SASL 警报事件处理器”一节中，我们了解了警报处理器（alarm handler），它是一个简单的警报管理器和处理程序，任何基于 OTP 的系统启动时默认都会启动它。SASL 还有一个非常基本的功能，即通过其 *overload* 库模块来调节系统中的 CPU 负载。当我们在第 13 章更详细地讨论负载调节机制，讨论到典型的 Erlang 节点的体系结构时会谈到这一点。

本章重点介绍的是 SASL 报告，使用它有助于在进程启动、终止和重新启动时监视监督树中的活动。你在本书前几章中碰到过 SASL 报告。主要是在启动 application、监督者和 worker 进程时在 shell 中看到的一些打印输出。你可能已经注意到，只有当 SASL

application 启动并且 `sasl_error_logger` 环境变量未设置为 `false` 时，它们才会出现。

SASL 会启动一个事件处理程序用于接受以下各类报告：

supervisor reports，监督者报告

当一个子进程异常终止时，由监督者发出。

progress reports，进度报告

当监督者启动或重启子进程时发出，以及当 application master 启动或重启 application 时发出。

error reports，错误报告

behavior 异常终止时由 behavior 自身发出。

crash reports，崩溃报告

由通过 `proc_lib` 库启动的进程发出，默认情况下包括 behavior。我们将在下一章中介绍 `proc_lib`。

默认设置是将报告打印到标准 I/O。可以通过设置环境变量来覆盖此选项，这些变量允许你将报告发送到环绕式（wraparound）二进制日志，并限制转发哪些报告。报告的格式取决于你正在运行的 OTP 发行包的版本。让我们来看看有哪些 SASL 环境变量可以帮助你控制报告。

`sasl_error_logger`

默认为 `tty` 并自动安装 `sasl_report_tty_h` 处理程序模块，该模块将报告打印到标准输出。如果你指定 `{file, FileName}`，其中 `FileName` 是包含文件相对路径或绝对路径的字符串，则会安装 `sasl_report_file_h` 处理程序，并将所有报告存储在 `FileName` 中。如果此环境变量被设置为 `false`，则不会安装任何处理程序，因此不会生成 SASL 报告。

`errlog_type`

可以取值为 `error`、`progress` 或 `all`，如果省略，默认为 `all`。使用此变量可限制由安装的处理程序所打印或记录到文件的错误报告或进度报告的类型。

`utc_log`

一个可选的环境变量，如果设置为 `true`，将把报告中的所有时间戳转换为通用协调时间（UTC）。

以下配置文件将所有 SASL 报告存储在一个名为 `SASLlogs` 的文本文件中。我们通过将 `sasl_error_logger` 环境变量设置为 `{file, "SASLlogs"}` 来实现。还可以使用 `utc_log`

环境变量启用 UTC 时间：

```
[{sasldb, [{sasldb_error_logger, {file, "SASLlogs"}},  
          {utc_log, true}}],  
 {bsc, [{frequencies, [1,2,3,4,5,6]}]}].
```

如果在本地非分布式节点中启动 *sasl* 和 *bsc* application, 你会发现所有日志作为纯文本存储在运行目录中。在我们的示例中, 我们仅显示第一个和最后一个报告。注意如何将 UTC 标记附加到时间戳上：

```
$ erl -pa bsc-1.0/ebin/ -config logtofile.config  
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> application:start(sasl), application:start(bsc).
```

```
ok
```

```
2> halt().
```

```
$ cat SASLlogs
```

```
=PROGRESS REPORT==== 9-Jan-2016::10:09:25 UTC ===
```

```
    supervisor: {local,sasl_safe_sup}
```

```
    started: [{pid,<0.40.0>},
```

```
              {name,alarm_handler},
```

```
              {mfargs,{alarm_handler,start_link,[]}},
```

```
              {restart_type,permanent},
```

```
              {shutdown,2000},
```

```
              {child_type,worker}]
```

```
...<snip>...
```

```
=PROGRESS REPORT==== 9-Jan-2016::10:09:33 UTC ===
```

```
    application: bsc
```

```
    started_at: nonode@nohost
```

文本文件在开发阶段可能会很好, 但是移至生产环境时, 最好将其改移到以可搜索的二进制格式存储的环绕型事件日志中。由于文本和二进制格式由不同的处理程序实现, 因此它们可以同时添加和并行运行。要安装二进制日志处理程序 `error_logger_mf_h`, 你必须设置三个环境变量。如果其中任何一个被禁用, 则不会添加处理程序。所需的环境变量是：

`error_logger_mf_dir`

指定存储二进制日志的目录的字符串。默认值为 `(".")`, 指定为当前工作目录。如果此环境变量设置为 `false`, 则不会安装处理程序。

`error_logger_mf_maxbytes`

定义每个日志文件的最大大小（以字节为单位）的整数。

`error_logger_mf_maxfiles`

1 到 256 之间的整数，指定生成的环绕日志文件的最大数量。

借助我们的 *bsc* 示例，让我们尝试使用本书代码库中的 *rb.config* 配置文件将 SASL 日志存储在二进制文件中。请注意，通过将 `sasl_error_logger` 环境变量设置为 `false`，我们显式关闭了发送事件到 shell，而且把 `frequencies` 设为原子 `crash` 而非整数列表，以故意造成尝试分配频率时进程出错：

```
[{sasl, [{sasl_error_logger, false},
        {error_logger_mf_dir, "."},
        {error_logger_mf_maxbytes, 20000},
        {error_logger_mf_maxfiles, 5}]},
{bsc, [{frequencies, crash}]}].
```

我们在 shell 命令 1 中启动 *bsc* application，并在 shell 命令 2 中触发频率服务器崩溃——使得程序尝试将原子 `crash` 模式匹配成 `frequency` 模块的 `allocate/2` 函数中的头和尾：

```
$ erl -pa bsc-1.0/ebin -config rb.config
```

```
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> application:start(sasl), application:start(bsc).
```

```
ok
```

```
2> frequency:allocate().
```

```
=ERROR REPORT==== 9-Jan-2016::19:24:30 ===
```

```
** Generic server frequency terminating
```

```
** Last message in was {allocate,<0.34.0>}
```

```
** When Server state == {data,[{"State",{available,crash},{allocated,[]}}]}
```

```
** Reason for termination ==
```

```
** {function_clause,[{frequency,allocate,
```

```
                        [{crash,[]},<0.34.0>],
```

```
                        [{file,"bsc-1.0/src/frequency.erl"},
```

```
                        {line,99}]}],
```

```
...<snip>...
```

```
3> rb:start().
```

```
rb: reading report...done.
```

```
{ok,<0.56.0>}
```

```
4> rb:list().
```

No	Type	Process	Date	Time
==	====	=====	=====	=====
14	progress	<0.37.0>	2016-01-09	19:24:26
13	progress	<0.37.0>	2016-01-09	19:24:26
12	progress	<0.37.0>	2016-01-09	19:24:26
11	progress	<0.37.0>	2016-01-09	19:24:26
10	progress	<0.24.0>	2016-01-09	19:24:26
9	progress	<0.46.0>	2016-01-09	19:24:26
8	progress	<0.46.0>	2016-01-09	19:24:26
7	progress	<0.46.0>	2016-01-09	19:24:26
6	progress	<0.24.0>	2016-01-09	19:24:26
5	error	<0.46.0>	2016-01-09	19:24:30
4	crash_report	frequency	2016-01-09	19:24:30
3	supervisor_report	<0.46.0>	2016-01-09	19:24:30
2	progress	<0.46.0>	2016-01-09	19:24:30
1	progress	<0.46.0>	2016-01-09	19:24:30

ok

在 shell 中自己尝试一下，这将有助于你了解 application 和监督树的工作原理。你会注意到的第一件事是，即使我们将 `sasl_error_logger` 设置为 `false`，仍然会收到一个错误报告。这是因为环境变量能控制的报告类型范围只包括 `supervisor`、`crash` 和 `progress` 三种。因此无论配置文件如何设置，错误报告总是会打印输出。考虑到我们的焦点是报表浏览器（report browser），因此我们在试运行中缩减了此特定错误报告的内容长度。

触发崩溃后，我们使用 shell 命令 3 中的 `rb:start()` 启动报告浏览器。待其读取所有报告后，我们通过 `rb:list()` 将它们列出。每当你想不起报告浏览器的某些命令时，使用 `rb:help()` 将会列出它们。进度报告 14-6（以逆序列出，最老的数字最大）是启动 application 及其监督树的进度报告。我们从检查报告 1-5 开始：

- 频率服务器由于其异常终止而产生报告 4 和 5。报告包含了事后调试和故障排除所需的补充信息。
- 报告 3 是由于终止而由监督者生成的。它包含监督者存储的该子进程的特定信息。
- 报告 1 和 2 是重启的子进程生成的。在我们的场景中，频率服务器崩溃，并且 `simple_phone_sup` 被终止，而由于顶级监督者 `bsc_sup` 的 `rest_for_all` 策略，使其重新启动。

236 进度报告

进度报告是由监督者启动子进程，包括 worker 进程或下级监督者时发出的。这些报告中包含监督者的名字和正在启动的子进程的规格。当启动或重新启动 application 时，application master 进程也会发出此报告。在这种情况下，报告显示的是 application 的名称和启动它的节点。这里有一个例子：


```
5> rb:show(6).
```

```
PROGRESS REPORT <0.7.0>
```

```
2016-01-09 19:24:26
```

```
=====
application
started_at
```

```
bsc
nonode@nohost
```

```
ok
```

示例中的进度报告告诉我们 *bsc* application 正确启动了。请注意我们是如何使用 *rb:show/1* 命令查看各个报告的。

错误报告

错误报告是 *behavior* 在异常终止时生成的。在我们的例子中，频率服务器在异常终止时生成报告。你可以调用 *error_logger:error_msg(String, Args)* 生成自己的错误报告，但是我建议你别这么做。谨慎使用此命令，只有出现意外错误时才用，因为用户生成的报告太多会隐藏真正严重的问题并使日志变得混乱，从而在查找崩溃报告和其他实际错误时更难找到重要的详细信息。以下是我们的示例中产生的一个错误报告：

```
6> rb:show(5).
```

```
ERROR REPORT <0.51.0>
```

```
2016-01-09 19:24:30
```

```
=====
** Generic server frequency terminating
** Last message in was {allocate,<0.34.0>}
** When Server state == {data,[{"State",{available,crash},{allocated,[]}]}}
** Reason for termination ==
** {function_clause,[{frequency,allocate,
                        [{crash,[],<0.34.0>},
                        [{file,"bsc-1.0/src/frequency.erl",
                          {line,99}}],
                        {frequency,handle_call,3,
                          [{file,"bsc-1.0/src/frequency.erl",
                            {line,66}}],
                        {gen_server,try_handle_call,4,
                          [{file,"gen_server.erl",{line,629}}],
                        {gen_server,handle_msg,5,
                          [{file,"gen_server.erl",{line,661}}],
                        {proc_lib,init_p_do_apply,3,
                          [{file,"proc_lib.erl",{line,240}}]}}]
```

```
ok
```

```
7> error_logger:error_msg("Error in ~w. Division by zero!~n", [self()]).
```

237

ok

```
=ERROR REPORT==== 9-Jan-2016::19:28:19 ===
Error in <0.57.0>. Division by zero!
```

崩溃报告

由 `proc_lib` 库启动的进程会产生崩溃报告。如果查看示例的退出原因，你将意识到这适用于所有从该库启动的 `behavior`。主 `behavior` 循环中的 `try-catch` 会捕获异常终止，并生成崩溃报告。如果 `behavior` 或进程以原因 `normal` 终止，或者当监督者终止 `behavior`，理由为 `shutdown` 时，则不会生成报告。崩溃报告包含关于崩溃进程的信息，包括退出原因、初始函数和消息队列，以及其他通常可以使用 `process_info` BIF 查看到的进程信息。我们的示例中的崩溃报告如下所示：

```
8> rb:show(4).
```

```
CRASH REPORT <0.51.0> 2016-01-09 19:24:30
```

```
=====
Crashing process
```

```
  initial_call      {frequency,init,['Argument__1']}
    pid            <0.51.0>
  registered_name    frequency
    error_info
      {exit,
```

```
        {function_clause,
          [{frequency,allocate,
            [{crash,[],<0.34.0>},
             [{file,"bsc-1.0/src/frequency.erl",{line,99}}],
            {frequency,handle_call,3,
              [{file,"bsc-1.0/src/frequency.erl",{line,66}}],
            {gen_server,try_handle_call,4,
              [{file,"gen_server.erl",{line,629}}],
            {gen_server,handle_msg,5,
              [{file,"gen_server.erl",{line,661}}],
            {proc_lib,init_p_do_apply,3,
              [{file,"proc_lib.erl",{line,240}}]}],
          [{gen_server,terminate,7,[{file,"gen_server.erl",{line,826}}],
            {proc_lib,init_p_do_apply,3,
              [{file,"proc_lib.erl",{line,240}}]}]}]}
```

```
  ancestors      [bsc,<0.47.0>]
  messages       []
  links          [<0.48.0>]
  dictionary     []
  trap_exit      false
```



```

status running
heap_size 987
stack_size 27
reductions 412

```

ok

监督者报告

监督者报告是由监督者检测到子进程异常终止后生成的。其通常包含子进程自己输出的错误信息。监督者报告中包含生成报告的监督者的名称和发生错误的子进程所处的阶段：

9> **rb:show(3).**

```
SUPERVISOR REPORT <0.48.0> 2016-01-09 19:24:30
```

```
=====
Reporting supervisor {local,bsc}
```

Child process

```
errorContext child_terminated
```

reason

```

{function_clause,
 [{frequency,allocate,
   [{crash,[],<0.34.0>},
   [{file,"bsc-1.0/src/frequency.erl"},{line,99}]}],
 {frequency,handle_call,3,
   [{file,"bsc-1.0/src/frequency.erl"},{line,66}]}],
 {gen_server,try_handle_call,4,
   [{file,"gen_server.erl"},{line,629}]}],
 {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,661}]}],
 {proc_lib,init_p_do_apply,3,
   [{file,"proc_lib.erl"},{line,240}]}}}

```

```
pid <0.51.0>
```

```
id frequency
```

```
mfargs {frequency,start_link,[]}
```

```
restart_type permanent
```

```
shutdown 2000
```

```
child_type worker
```

ok

如果你查看示例输出的顶部，会看到在发生错误时子进程处于的阶段：`start_error`、`child_terminated` 或 `shutdown_error` 之一。在本例中，进程是由于运行时错误而导致终止的，所以报告中展示的阶段为 `child_terminated`。后续内容则是终止原因和子进程规格。

239 你可以自己查看最后两个进度报告。它们是频率服务器和电话监督者重新启动时生成的进度报告。使用 `rb:help()` 命令，并花点时间来测试报告浏览器（report browser）中的各个命令，特别是过滤器和正则表达式。

SASL 日志会保释你

SASL 日志应当在所有生产环境节点上默认启用，因为它们将在你调查节点崩溃或尝试重新启动节点时成为你的第一个调查点。在大多数情况下，错误、崩溃和监督者报告已经包含了足够的信息能帮助你确定发生了什么。建议始终留有一个单独的启动脚本，允许你只启动 Erlang 自身（包含了 *sasl*），并使用 `rb:start([report_dir, Dir])` 命令加载日志，因为有可能碰到运行你的发行版的 Erlang 节点无法重新启动的状况。不要指望用你正在调查的 Erlang 节点读取它们，因为它很可能启动不了。如果你有外部报警和监控系统，一般而言每当收到错误、崩溃和监督者报告时就自动产生通知是一个好主意，这让你有机会能够及时调查。生产环境中往往运行着许多节点——有可能数以千计，因此把这些通知聚合在一起可以让你的工作变得轻松一点。你可以通过编写自己的事件处理程序并将其挂接到 SASL 事件管理器从而轻松地将其转发给第三方工具。

总结

在本章中，我们介绍了 application 行为模式，它使得我们能够将代码、资源、配置文件和监督树打包为一个整体。application 是你的系统中可重复使用的构建块，它们作为单个单元被加载、启动和停止。它们提供了许多功能，例如分布式集群中的分阶段启动、同步、故障转移等，以及基本的监控和日志记录服务。

表 9-1 列出了用于控制 application 的主要函数。

表 9-1: application 回调函数

application 函数或动作	application 回调函数
<code>application:start/1</code> , <code>application: start/2</code>	<code>Module:start/2</code> , <code>Module:start_phase/3</code>
<code>application:stop/1</code>	<code>Module:prep_stop/1</code> , <code>Module:stop/2</code>

你可以在 application 手册页中了解更多相关信息（<http://bit.ly/erlang-app>），关于资源文件方面的内容则可以在 app 手册页中了解（<http://erlang.org/doc/man/app.html>）。“OTP 设计原则用户指南”（OTP Design Principles User’s Guide）附带了标准的 Erlang 文档，其中分别介绍了一般型（normal）、内含型（included）和分布式（distributed）application。要了解有关我们提到的工具的更多信息，它们也有各自的手册页，例如报

240

告浏览器 (report browser) rb 的手册页 (<http://bit.ly/erlang-rb>) 以及 observer 的手册页 (<http://yaws.hyber.org/>)。阅读本章中提供的示例代码, 可了解 Erlang 发行版中的 application 是如何打包和配置的。

接下来是什么

现在我们已经明白如何创建 application 了, 它是 Erlang 系统的基本构件块, 接下来我们将看看如何将它们组合在一起构成发行包 (release), 并使用引导文件启动系统。但在此之前, 我们先来学习一些用于实现特殊进程 (special process) 的库, 并使用这些知识来定义自己的行为模式。我仿佛听到你问什么是特殊进程? 其实它们依然是进程, 但它们不属于 `stdlib` application 自带的 OTP 行为模式, 却可以添加到 OTP 监督树中。继续阅读, 你会了解更多。

基于特殊进程打造自己的behavior

就绝大多数情况而言，OTP behavior 提供的并发设计模式都能满足你的项目所需。但是，在某些情况下，你可能希望将非标准的自制 behavior 进程添加到标准监督树上，同时保证最终的 application 依然兼容 OTP 体系。例如，已有 behavior 的性能达不到你的需求——它们对通用部分和错误处理相分离的设计增加的抽象分层影响了性能。于是你实现了自己的方案，并且成功地将代码分离为通用模块和专用模块，接下来你就需要编写新的 behavior 了。或者你可能想要做一些简单的事情，比如将纯 Erlang 进程添加到监督树，从而使你的 OTP 发行包具备比监督者桥接器（supervision bridge）更广阔的兼容性。例如，你可能不得不把一些早期刚开始探索 Erlang 时写的（现如今看来其实算不上好的）概念验证性代码封装后放到生产环境中去。^{注1}

这种能够添加到 OTP 监督树并打包进 application 的进程，我们称之为特殊进程（special process）。本章将介绍如何编写自己的特殊进程，它一方面具备纯 Erlang 的灵活性，同时又兼具 OTP 的各种优点。我们还将进一步介绍如何将你的特殊进程转换为 OTP behavior，方法是将代码拆分为通用模块和专用模块，并基于约定的回调函数将它们相互连接。如果你没有设计自己的 behavior 的需要，或者对此类幕后原理毫无兴趣，尽可以放心地跳到下一章。等你有这方面需求时可以随时回来阅读。反之，倘若我们激起了你的好奇心，
242 请继续阅读吧。

特殊进程

使一个进程被视为特殊进程，进而能够成为 OTP 监督树的一部分，它必须：

- 使用 `proc_lib` 模块启动并链接到它的父进程。

注1 对于那些在大公司工作的人来说，我们说的就是那种花了大量时间开会讨论只为获得移植到 OTP 的批准的项目，实际上有这么多时间，代码早就可以重构好了。

- 能够处理系统消息、系统事件和关闭请求。
- 如果含有动态模块，则要能返回模块列表，就像我们在事件管理器中定义子进程的规格时一样。

虽然是可选的，但是如果进程也能够处理调试标志并生成跟踪消息，那么会很有用。

我们将通过实现一个互斥体——一种用于将资源访问顺序化的工具——作为示例来向你展示实现特殊进程的方法。

互斥体

互斥体如其名所示，表示相互排斥。它能确保一次只允许一个进程在临界区内执行代码。临界区内的资源可能是打印机、共享内存或任何其他请求必须被序列化的设备——原因是此类设备一次只能处理一个客户端。一个进程如果正执行访问这些资源的代码，我们称其为位于临界区内。其他进程要想进入临界区，必须等待它执行完临界区内的全部代码并退出后才行。

在 Erlang 中，程序员可以将一个互斥体实现为一个有限状态机 (FSM)，借助一个进程来使得并发的客户端请求顺序化，并使用邮箱和选择性接收来管理请求队列。因为我们正在实现一个 FSM，所以应该问问自己为什么不使用 `gen_fsm` 这一已有的 `behavior` 模块。原因是此 `behavior` 以及所有其他标准的 OTP `behavior` 都不允许我们选择性地通过模式匹配来接收消息。相反，标准的 `behavior` 迫使我们按照事件到达的顺序处理事件。相比之下，通过使用进程邮箱和选择性接收来管理等待互斥体的客户端进程的队列，我们简化了代码，因为一次只能处理一个客户请求，而不必担心其他人在等待队列。

互斥体是有两个状态的有限状态机 (FSM)，*free* 状态和 *busy* 状态。任何一个客户端如果想进入临界区都需要调用函数 `mutex:wait(Name)` 才能做到，其中 `Name` 是一个变量，对应目标互斥体的注册名称。`wait` 调用是同步的，只有调用进程被允许进入临界区时才会返回。当发生这种情况时，有限状态机转换到状态 *busy*。

◀ 243

请求被存储在邮箱中，并按先进先出的原则处理。如果互斥体此时处于状态 *busy*——即正被另一个进程阻塞，则请求将留在邮箱中直到互斥体返回状态 *free* 时再处理。在当前占用进程准备离开临界区时，它会调用 `mutex:signal(Name)`，这是一个异步调用，作用是释放互斥体。当发生这种情况时，有限状态机转换回状态 *free*，准备好处理下一个请求。图 10-1 显示了互斥体的状态转换。

让我们来看看 `mutex` 模块，先从客户端函数开始（其他导出函数会在稍后定义）：

```
-module(mutex).
```

```
-export([start_link/1, start_link/2, init/3, stop/1]).  
-export([wait/1, signal/1]).
```

```
wait(Name) ->  
    Name ! {wait,self()},  
    Mutex = whereis(Name),  
    receive  
        {Mutex,ok} -> ok  
    end.
```

```
signal(Name) ->  
    Name ! {signal,self()},  
    ok.
```

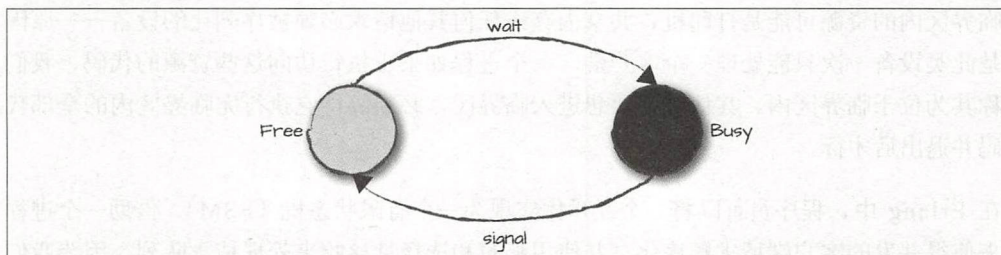


图 10-1: 互斥体内的状态转换

标准 OTP behavior 中妥善处理了许多边界状况，以符合许多程序员的自然直觉。在查看 `gen_server` 或 `gen_fsm` 模块中的代码时，你可能已经亲眼见过了那些代码。但是，在实现特殊进程时，你需要自行决定处理哪些边界状况，并自行实现处理过程。在我们的例子中，为了简明，并没有做任何此类处理。但为了让你意识到我们所说的问题，请回头看一下 `wait/1` 函数，我们没有检查 `Name` 是否存在，也没有监视互斥体是否终止，而这种状况可能在客户端进程停留在 `receive` 子句中时发生。我们也不处理互斥体在 `whereis/1` 之前终止并且立即重新启动并重新注册的情况，这将导致 `wait/1` 停留在 `receive` 子句中等待永远收不到的消息。对于互斥进程死锁或挂起的情形，我们也没有设计任何超时处理。

启动特殊进程

启动特殊进程时，请使用 `proc_lib` 库模块中定义的 `start` 和 `spawn` 函数，而不是 Erlang 中标准的 `spawn` 和 `spawn_link` BIF。`proc_lib` 函数会在进程字典中存储进程的名称 (`name`)、标识 (`identity`)、父进程 (`parent`)、祖先 (`ancestors`) 以及初始函数调用。如果进程异常终止，将生成 SASL 崩溃报告并将其转发给错误日志记录器。其中包含启动时存储的全部与进程相关的信息，以及终止原因。并且和其他 behavior 一样，还提供了



一些功能让你可以在初始化阶段（init phase）做同步启动。

注意，一种常见的错误是把一个进程附加到监督树上，但该进程却并没有实现 behavior。犯此错误时不会出现编译时或运行时警告，因为监督者仅仅检查其是否返回元组 {ok, Pid}。并且针对 Pid 也并不做什么检查。一直要等到崩溃、重新启动或升级时，才会发现问题出在哪里。而且由于这些进程并不遵循标准的 behavior，除非你已经测试了重启策略，否则与常规和有序的故障排除相比，定位此类问题着实令人劳心费神。对于不遵循 OTP 标准的进程，请使用第 8 章“监督者桥接器”一节中介绍的方案。而本章我们将向你介绍如何创建一个遵循 OTP 标准的进程。

用于启动特殊进程的基础模板

启动特殊进程时，建议使用 `proc_lib:start_link(Mod, Fun, Args)` 方法而不是 `spawn_link/3 BIF`。它会根据你提供的模块、函数和参数列表同步分裂出一个进程，并等待这个新进程调用 `proc_lib:init_ack(Value)` 表明自己已正确启动。然后其中的 Value 会被发送回父进程，成为 `start_link/3` 调用的返回值。请注意我们是如何在 `start_link` 调用中传递可选的 `DbgOpts` 调试选项参数的。我们在第 5 章中介绍过。现在，假设 `DbgOpts` 是一个空的列表。另外请留意，我们是如何将 Parent 进程的 ID 传递给 `init/3` 函数的。因为我们的主循环需要这个值，而该值来自调用 `start_link/2` 时执行 `self()` BIF 获得的结果。

```
start_link(Name) ->
    start_link(Name, []).

start_link(Name, DbgOpts) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, DbgOpts]).

stop(Name) -> Name ! stop.

init(Parent, Name, DbgOpts) ->
    register(Name, self()),
    process_flag(trap_exit, true),
    Debug = sys:debug_options(DbgOpts),
    proc_lib:init_ack({ok,self()}),
    free(Name, Parent, Debug).
```

245

当初始化进程状态时，我们先将互斥体以 Name 为名称进行注册。我们设置了 `trap_exit` 标志，以便可以接收来自链接集（linked set）中的进程的退出信号（如果互斥体失败，我们需要通知或终止调用者，所以此处使用的是链接而不是监视器）。最后，我们使用 `sys:debug_options(DbgOpts)` 调用初始化调试跟踪标志。`debug_options/1` 的返回值传递并存储在进程状态中用作循环数据。当我们的特殊进程需要生成跟踪消息，或是接收



到系统消息需要更新其跟踪标志时，就会需要用到此值。

如图 10-2 所示，一旦状态被初始化，我们就调用 `proc_lib:init_ack(Value)` 来通知父进程此特殊进程已经正确启动。`Value` 会被送回，成为 `proc_lib:start_link/3` 调用的返回值。此处我们通常返回 `{ok, self()}`，这并非强制做法，而是考虑到了监督者希望其子进程的启动函数返回 `{ok, Pid}` 这一点。如果初始化过程中在调用 `init_ack/1` 之前任何地方出现错误，那么 `proc_lib:start_link/3` 将以对应的错误原因终止。看看 `init/3` 函数的最后一行，请注意，调用 `free` 函数用来将有限状态机指向第一个状态，而 `Name`、`Parent` 和 `Debug` 则对应的是进程状态，请区分开。

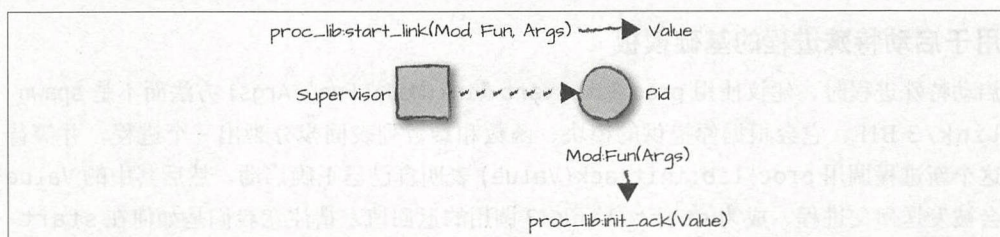


图 10-2: 启动特殊进程

可以用来同步启动一个特殊进程的调用有：

```
proc_lib:start(Module, Function, Args)
proc_lib:start(Module, Function, Args, Time)
proc_lib:start(Module, Function, Args, Time, SpawnOpts) -> Ret
proc_lib:start_link(Module, Function, Args)
proc_lib:start_link(Module, Function, Args, Time)
proc_lib:start_link(Module, Function, Args, Time, SpawnOpts) -> Ret
proc_lib:init_ack(Ret)
proc_lib:init_ack(Parent, Ret) -> ok
```

`start/3,4,5` 和 `start_link/3,4,5` 函数的返回值 `Ret` 都来自 `init_ack/1,2` 调用。与其他 `behavior` 一样，`SpawnOpts` 是一个列表，其中包含 `spawn` BIF 接收的所有选项，但排除与 `monitor` 有关的选项。如果在 `Time` 毫秒内未调用 `init_ack`，则启动函数将返回 `{error, timeout}`。如果使用 `spawn` 或 `spawn_opt`，请不要忘记通过 `link/1` BIF 或通过 `SpawnOpts` 中的 `link` 选项将子进程链接到父进程。

异步启动特殊进程

这些是标准 `spawn` 和 `spawn_link` 函数的变体，用于需要异步启动的情形，例如需要同时启动数百个新进程时。这些函数会创建子进程，并立即返回子进程的 `PID`：

```
proc_lib:spawn(Fun)
```




```

proc_lib:spawn_link(Fun)
proc_lib:spawn_opt(Fun, SpawnOpts) -> Pid
proc_lib:spawn(Module, Function, Args)
proc_lib:spawn_link(Module, Function, Args)
proc_lib:spawn_opt(Node, Function, SpawnOpts) -> Pid

```

其他同步启动特殊服务器的方式还包括基于函数的方式，以及使用 `SpawnOpts` 选项的方式。

使用异步分裂 (asynchronous spawning) 时要小心，因为这些函数可能会导致多个进程并行运行，触发竞态条件，导致程序的不确定性。在讨论通用服务器 (`gen_server`) 时，我们曾在第 4 章“启动一个 server”一节中提及过的论点在此处同样有效。如果一种启动错误与以特定顺序发生的一定数量的并发事件有关，则该启动错误会很难重现，这一问题在多核架构中会愈发严重。因此，为了能够确定性地重现启动错误，请同步创建你的进程。

不管如何启动你的特殊进程（同步或异步），必须将它们与父进程（默认情况下即监督者进程）相链接。如果你使用 `start_link`、`spawn_link` 或通过 `SpawnOpts` 传递 `link` 选项的方式，那么这个操作会自动执行。但是请小心，由于此处实际上并无检查去确保该进程与监督者是否真的成功链接，所以即使在这里，倘若出现此类疏漏也将很难排除和检测。

互斥体的状态

247

正如我们所看到的，一个互斥体有两个状态，*free* 和 *busy*，而且它们是以尾递归函数的形式实现的。同步的 `wait` 事件和异步的 `signal` 事件以消息的形式连同客户端 `pid` 一起发过来。状态和事件的不同组合决定了不同的动作和状态转换。请注意，在 *free* 状态下，我们只接受 `wait` 事件，通过消息 `{self(),ok}` 通知客户端允许进入临界区。然后互斥体将转换到 *busy* 状态，其中唯一做了模式匹配的事件是 `signal`，且必须由 `Pid` 进程发过来。你应该已经注意到了函数头部绑定了 `Pid`，对应持有互斥锁的客户端进程。在收到 `signal` 事件后，互斥体将转换回 *free* 状态：

```

free(Name, Parent, Debug) ->
  receive
    {wait,Pid} ->
      Pid ! {self(),ok},
      busy(Pid, Name, Parent, Debug);
    stop ->
      ok
  end.

```



```

busy(Pid, Name, Parent, Debug) ->
    receive
        {signal,Pid} ->
            free(Name, Parent, Debug)
    end.

```

请注意，只有当互斥体处于 *free* 状态时，我们才接收 *stop* 消息。反之，倘若允许在 *busy* 状态中停止互斥锁，则客户端此时在其临界区中执行的代码将可能被放任而产生难以预料的后果，有可能导致数据损坏，因为该互斥锁可能又会被其他客户端进程重新启动并进入。通过限制仅允许 *free* 状态下停止互斥体，可以保证关闭过程清晰有序。

到现在为止一切都还不错。接下来除了有限状态机 (FSM) 的知识外，我们还会用到一些先前介绍过的 Erlang 基础知识。现在让我们开始扩展状态来处理其他一些特殊进程被要求必须处理的系统消息吧。

处理退出

如果特殊进程的父进程终止，那么特殊进程也必须终止。如果你的进程没有捕获退出信号，那么运行时会处理这一细节，因为你应当早已将其链接到父进程了。非 *normal* 的退出信号会传播给链接集中的所有进程，并以相同的原因终止它们。而因为 *normal* 的退出信号不会传播，但在 OTP 中，监督者保证了任何父进程都永远不会以此原因终止，所以你不必担心。

248 捕捉退出信号的特殊进程必须监督其父进程，因为它们可能会收到如下格式的信息：

```
{'EXIT', Parent, Reason}
```

其中 *Parent* 是父进程 *pid*，*Reason* 是终止原因。如果收到了这样的消息，那么这些特殊进程接下来应该做清理工作，具体而言可能在它们的 *terminate* 函数或是其他清理函数中去做，然后应该调用 *exit(Reason) BIF*。

在之前的例子中，互斥体设置了捕捉退出信号，所以我们必须监视父进程的终止。让我们扩展状态函数，通过调用 *terminate/2* 来处理父进程中的 *EXIT* 消息吧。除此之外，收到 *stop* 消息时我们也会调用 *terminate/2*。如果父进程终止时当前处于 *busy* 状态，我们会在调用 *terminate/2* 之前先终止持有该互斥体的进程：

```

free(Name, Parent, Debug) ->
    receive
        {wait,Pid} ->
            link(Pid),
            Pid ! {self(),ok},
            busy(Pid, Name, Parent, Debug);

```




```

stop ->
    terminate(shutdown, Name);
{'EXIT', Parent, Reason} ->
    terminate(Reason, Name)
end.

```

```

busy(Pid, Name, Parent, Debug) ->
    receive
        {signal, Pid} ->
            free(Name, Parent, Debug);
        {'EXIT', Parent, Reason} ->
            exit(Pid, Reason),
            terminate(Reason, Name)
    end.

```

```

terminate(Reason, Name) ->
    unregister(Name),
    terminate(Reason).

```

```

terminate(Reason) ->
    receive
        {wait, Pid} ->
            exit(Pid, Reason),
            terminate(Reason)
    after 0 ->
        exit(Reason)
    end.

```

terminate/2 做的第一件事就是取消互斥体先前所做的名称注册，这将确保任何试图发送到它的进程都以 badarg 原因终止。互斥体紧接着遍历整个邮箱，提取出 wait 请求来终止队列中所有等待中的进程。完成这些操作，互斥体可以确定已不再有因自己而挂起的客户端进程存在了，因此以 Reason 为理由终止了自己。

◀ 249

系统消息

除了监视父进程之外，特殊进程还需要管理以下格式的系统消息：

```
{system, From, Msg}
```

其中 From 是请求始发者，而 Msg 是系统消息内容。这样的系统消息可能来自监督者，目的是在软件升级期间挂起和恢复进程，或者来自某个使用了 sys 模块、意在操作或查询跟踪输出的客户端进程。到底消息来自谁，开发人员无须在意，因为对你而言它们只不过是某种不透明的数据类型，你只需将其传递即可。

不管请求是什么，这些调用都在 sys:handle_system_message(Msg, From, Parent,



Mod, Dbg, Data) 函数内部被处理, 如图 10-3 所示。这个调用的参数虽多, 但含义直白:

- Msg 和 From 来自系统消息。
- Parent 是父进程 pid, 在分裂特殊进程时传递过来的值。
- Mod 是实现特殊进程的模块的名称。
- Dbg 是调试数据, 最初由 sys:debug_options/1 调用返回。
- Data 指进程存储的循环数据。

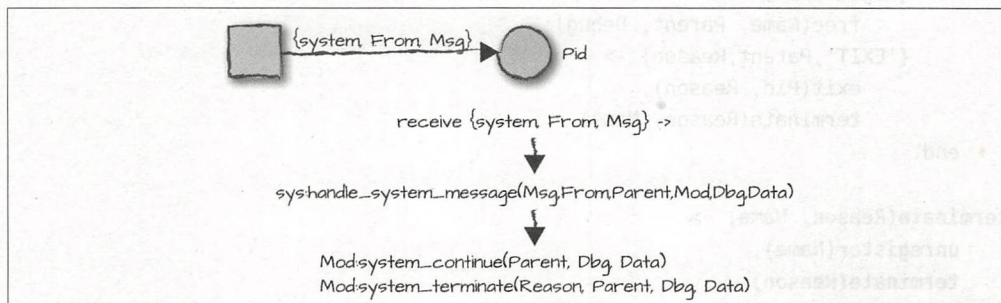


图 10-3: 处理系统消息

特殊进程模块中发起此调用的函数自身必须写成尾递归形式, 原因是此调用永远不会返回。反之如果每次收到系统消息时, 不进行尾递归将会导致内存泄漏。最后, 通过调用以下回调函数之一, 控制流将会回归给 Mod 模块中的特殊进程:

250 `Mod:system_continue(Parent, Debug, Data)`
`Mod:system_terminate(Reason, Parent, Debug, Data)`

如果控制流是通过回调函数 `system_continue/3` 返回的, 则你的特殊进程需要返回其主循环。而如果被调用的是 `system_terminate/4`——这也许是由于父进程下达关闭命令而导致的——则特殊进程接下来需要执行清理工作, 并以 Reason 原因终止。我们在互斥体的例子里会向你展示这些, 但目前让我们先了解清楚调试输出是如何运作的。

跟踪与日志事件

在本章前面介绍启动函数时, 我们讨论了 `SpawnOpts` 参数, 它允许将调试标志连带其他选项一起传递给特殊进程。在我们的 `mutex:start_link/2` 调用中, 可以在第二个参数中传递这些调试选项, 将它们绑定到 `DbgOpts` 变量。 `DbgOpts` 中可包含零个或多个 `trace`、`log`、`statistics` 和 `{log_to_file, FileName}` 等第 5 章曾介绍过的标志。该列表由特殊进程传递给 `sys:debug_options(DbgOpts)` 调用, 该调用会创建一些调试例程 (debug routines)。无法识别或不支持的调试选项将被忽略。在我们的例子中, 存储在变量 `Debug` 中的调用的返回值保存在传递给所有系统调用的特殊进程循环数据中。还记



得第 5 章“跟踪与记录”一节中的示例吗，我们在运行时打开或关闭了跟踪，将其打印到 shell 中并将其转移到文件中？如果所有内容都已正确初始化，则可以使用特殊进程生成类似的跟踪日志，并在运行时打开和关闭这些选项。所有来自诸如 `sys:trace/3` 或 `sys:log/2` 等调用的请求，均作为系统消息接收和处理。在各次调用之间，Debug 列表的内容是可能改变的，并作为 `system_continue/3` 回调函数的一部分返回。

生成跟踪事件是很简单的操作，通过调用此函数即可完成：

```
sys:handle_debug(Debug, DbgFun, Extra, Event)
```

其中：

- Debug 是一个已经初始化好的调试选项。
- DbgFun 是一个三元函数，用于格式化跟踪事件。
- Extra 是某种格式化事件时会用到的数据，通常是进程名称或循环数据之类。
- Event 正是你想要打印的跟踪事件。

DbgFun 是一个用于格式化事件的函数，其内部有时会调用其他函数。该函数会接收 sys 模块传递给它的一些参数，包括要写入的 I/O 设备是哪一个，值可以是原子 `standard_io` 或 `standard_error`，也可以是 `file:open` 调用返回的 `pid`。Extra 和 Event 来自调用 `handle_debug/4` 时传入的参数：

```
fun(Dev, Extra, Event) ->
  io:format(Dev, "mutex ~w: ~w~n", [Extra,Event])
end
```

通过调用 `sys:install/2`，你还可以在运行时添加自己的跟踪函数，并通过在跟踪函数头部使用模式匹配检查事件进而决定执行流。介绍了系统消息和跟踪输出后，我们把它们添加到互斥体例子里来进一步了解它们是如何组合在一起的。

合在一起

为了方便阅读，我们把整个互斥体例子的代码都放在了一起。请注意其中我们是如何扩展 *free* 和 *busy* 状态以涵盖跟踪消息和系统消息的。让我们聚焦于其功能，从跟踪消息部分开始。

当收到 `wait` 和 `signal` 事件时，我们调用 `sys:handle_debug(Debug, fun debug/3, Name, Event)`，其中 Event 是 `{wait,Pid}` 或 `{signal,Pid}`。这个调用导致控制权移交给 sys 模块，进而最终调用了我们的调试函数。具体到我们的例子而言，就是本地函数 `debug/3`。看看此函数，特别要注意其中是如何使用 I/O 设备，以及传递给它的事件及额外参数的。`handle_debug/4` 返回 `NewDebug`，它作为参数传递给下一个状态。在研读此例时，

◀ 251



请记住互斥体进程本身并不实现它所保护的服务。它只是充当一种信号，控制其他进程去访问受保护的服务。完整的互斥体示例如下所示：

```
-module(mutex).

-export([start_link/1, start_link/2, init/3, stop/1]).
-export([wait/1, signal/1]).
-export([system_continue/3, system_terminate/4]).

wait(Name) ->
    Name ! {wait,self()},
    Mutex = whereis(Name),
    receive
        {Mutex,ok} -> ok
    end.

signal(Name) ->
    Name ! {signal,self()},
    ok.

start_link(Name) ->
    start_link(Name, []).

start_link(Name, DbgOpts) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, DbgOpts]).

stop(Name) -> Name ! stop.

init(Parent, Name, DbgOpts) ->
    register(Name, self()),
    process_flag(trap_exit, true),
    Debug = sys:debug_options(DbgOpts),
    proc_lib:init_ack({ok,self()}),
    NewDebug = sys:handle_debug(Debug, fun debug/3, Name, init),
    free(Name, Parent, NewDebug).

free(Name, Parent, Debug) ->
    receive
        {wait,Pid} ->           %% The user requests.
            NewDebug = sys:handle_debug(Debug, fun debug/3, Name, {wait,Pid}),
            Pid ! {self(),ok},
            busy(Pid, Name, Parent, NewDebug);
        {system,From,Msg} ->    %% The system messages.
            sys:handle_system_msg(Msg, From, Parent,
                                   ?MODULE, Debug, {free, Name});
```



```

stop ->
    terminate(stopped, Name, Debug);
    {'EXIT',Parent,Reason} ->
        terminate(Reason, Name, Debug)
end.

busy(Pid, Name, Parent, Debug) ->
    receive
        {signal,Pid} ->
            NewDebug = sys:handle_debug(Debug, fun debug/3, Name, {signal,Pid}),
            free(Name, Parent, NewDebug);
        {system,From,Msg} ->    %% The system messages.
            sys:handle_system_msg(Msg, From, Parent,
                ?MODULE, Debug, {busy,Name,Pid});
        {'EXIT',Parent,Reason} ->
            exit(Pid, Reason),
            terminate(Reason, Name, Debug)
    end.

debug(Dev, Event, Name) ->
    io:format(Dev, "mutex ~w: ~w~n", [Name,Event]).

system_continue(Parent, Debug, {busy,Name,Pid}) ->
    busy(Pid, Name, Parent, Debug);
system_continue(Parent, Debug, {free,Name}) ->
    free(Name, Parent, Debug).

system_terminate(Reason, _Parent, Debug, {busy,Name,Pid}) ->
    exit(Pid, Reason),
    terminate(Reason, Name, Debug);
system_terminate(Reason, _Parent, Debug, {free,Name}) ->
    terminate(Reason, Name, Debug).

terminate(Reason, Name, Debug) ->
    unregister(Name),
    sys:handle_debug(Debug, fun debug/3, Name, {terminate, Reason}),
    terminate(Reason).
terminate(Reason) ->
    receive
        {wait,Pid} ->
            exit(Pid, Reason),
            terminate(Reason)
    after 0 ->
        exit(Reason)
end.

```

当 `free` 和 `busy` 函数收到 `{system,From,Msg}` 时，它们递归调用 `sys:handle_system_msg(Msg, From, Parent, ?MODULE, Debug, {State, LoopData})`，将控制交给 `sys` 模块。系统消息的处理过程是在幕后进行的，然后才通过调用 `mutex` 模块中的 `system_continue/3` 或 `system_terminate/4` 来返回。如果这两个函数你没有做尾递归，就像前面提到的那样，将导致每收到一条系统消息都会产生一点内存泄漏。

在我们的例子中，如果 `system_continue` 被调用，我们所做的仅仅是调用并返回当前所对应的那个状态函数而已，区分当前处于哪个状态是根据 `free` 状态时循环数据包含的是 `Name` 而 `busy` 状态时是 `{Name, Pid}` 这一点，无论调用哪个状态函数，其中都会等待下一个事件或系统调用。而如果是 `system_terminate` 被调用的情况下，我们判断如果当前处于 `busy` 状态，则主动终止持有该互斥体的进程（当然这可能使系统状态变得不一致），然后调用 `terminate/2`。而如果当前是 `free` 状态，我们只需调用 `terminate/2` 即可。在这两种情况下，为了确保继续或终止时采取的是正确的动作，我们都应用了模式匹配。

系统消息和调试选项是在你的特殊进程中直接处理的。你只需重用这个例子中的代码即可，注意当控制还回给你时，要使用尾递归函数的方式回到你的循环函数或者状态函数中去。在查看互斥体的试运行之前，请再次阅读代码，并确保你理解了特殊进程是什么，为什么我们需要它以及我们实现它所采取的方法。

在试运行过程中，我们为特殊进程创建了一个子进程规格，然后在监督者 `mutex_sup` 中作为动态子进程启动。在这个例子中，我们没有展示监督者部分的代码，因为并无新意，仅仅是一些样板代码。其 `init/1` 函数所做的仅仅是返回一个监督者规范，其中指定使用 `one_for_one` 重启策略，并且每小时最多重新启动五次，以及一个空的子进程列表。这些源代码可以在本书的 GitHub 仓库中找到。

请留心在 `mutex:start_link/2` 参数中的子进程规格部分，我们是如何打开跟踪标志的。

254 这使得在 `shell` 命令 3 中启动互斥体时，打印了跟踪输出。在 `shell` 命令 4 和 5 中我们使用 `sys` 模块打开了其他调试选项：

```
1> ChildSpec = {mutex, {mutex, start_link, [printer, [trace]],
                       transient, 5000, worker, [mutex]}.
{mutex, {mutex, start_link, [printer, [trace]],
        transient, 5000, worker,
        [mutex]}}
2> mutex_sup:start_link().
{ok, <0.35.0>}
3> supervisor:start_child(mutex_sup, ChildSpec).
mutex printer: init
{ok, <0.37.0>}
4> sys:log(printer, {true, 10}).
```



```

ok
5> sys:statistics(printer, true).
ok
6> mutex:wait(printer), mutex:signal(printer).
mutex printer: {wait,<0.32.0>}
mutex printer: {signal,<0.32.0>}
ok
7> sys:log(printer, get).
{ok, [{wait,<0.32.0>},printer,#Fun<mutex.1.94496536>},
      [{signal,<0.32.0>},printer,#Fun<mutex.2.94496536>]]}
8> sys:log(printer, print).
mutex printer: {wait,<0.32.0>}
mutex printer: {signal,<0.32.0>}
ok
9> sys:get_status(printer).
{status,<0.37.0>,
  {module,mutex},
  [[{'$ancestors',[mutex_sup,<0.32.0>]],
    {'$initial_call',{mutex,init,3}}],
   running,<0.35.0>,
   [{statistics,{{2014,1,6},{8,50,36}},{reductions,66},0,0}},
    {log,{10,
          [{signal,<0.32.0>},printer,#Fun<mutex.2.94496536>},
           {wait,<0.32.0>},printer,#Fun<mutex.1.94496536>]]}},
    {trace,true}},
  {free,printer}}]
10> exit(whereis(printer), kill).
mutex printer: init
true
11> exit(whereis(mutex_sup), shutdown).
mutex printer: {terminate,shutdown}
** exception exit: shutdown

```

在 shell 命令 6 中，我们等待互斥体，然后发出信号将其释放，这个过程里的每个请求都会产生一个跟踪事件。在 shell 命令 7、8 和 9 中，我们通过 sys 模块获取一些跟踪和状态信息，随后进行一些测试，并在 shell 命令 10 和 11 中终止。

你自己可以做一些测试，测试多个客户端的情况，试试 SASL 报告浏览器以及其他 sys 模块中的诸如挂起和重新启动模块等命令。

动态模块和休眠

回顾第 8 章你也许还记得，我们需要在子进程规格中提供一个实现了 behavior 的模块列表，它们被用于判定在软件升级期间哪些进程应当被挂起。但在有些情况下，我们无法

在编译阶段就明确此模块列表，例如事件管理器和事件处理器。在监督者子进程规格的模块列表中，这些 behavior 被标记为原子 dynamic。同样的，特殊进程也一样可以有动态模块。

如果你的特殊进程模块在子进程规格中被标记为 dynamic，则如图 10-4 所示，你需要处理系统消息 {get_modules, From}。From 是监督者的 pid，在 From!{modules, ModuleList} 表达式中用于返回模块列表。

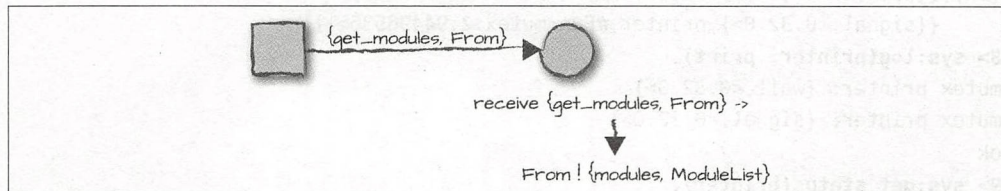


图 10-4: 获取动态模块

如果需要休眠你的特殊进程，别用 BIF，而应当使用：

```
proc_lib:hibernate(Mod, Fun, Args)
```

它与 BIF 以及标准 OTP behavior 的返回值一样具有使进程休眠的功能，但是它提供了一个额外的附加功能，它能够确保在进程唤醒时日志记录和调试功能仍然可用。

属于你自己的 behavior

现在你已经理解特殊进程方面的知识了，接下来我们进一步从概念上将其分解成通用部分和专用部分，从而实现属于我们自己的 behavior。当你遇到不同的几个进程需要遵循相似的模式工作，但现有的 OTP behavior 又不便于表达时，你就会想实现自己的 behavior。但是通用服务器 (gen_server)、通用有限状态机 (gen_fsm) 和通用事件管理器 (gen_event) 实际上足以满足大多数程序员的需求，因此请不要兴奋过度，在每个项目中都编写新的 behavior。很多时候你花时间过度开发的一个解决方案，实际上抽象为一个库模块就可以很容易地解决。

话虽如此，有时候确实有充分的理由来实现自己的 behavior。模式可以被抽象为通用部分和专用部分，特别是当通用部分有一定规模时这样做很有意义。如果沿着条路走下去，大多数时候你可以借助 gen_server 实现你的 behavior (或库)。如果这样做不可行，或是你考虑到性能开销需要避免使用 gen_server，那么实现时请确保你的 behavior 遵循了特殊进程使用 sys 和 proc_lib 模块时相关的设计规则。



如果你正在进入软件考古领域并对软件演化感兴趣，不妨看看早期 Erlang/OTP 发行包里的源代码。浏览旧的 behavior 代码，你会发现大部分 behavior 是在 `gen_server` 之上构建的。而如今的 OTP behavior，包括 `gen_server` 在内，是基于名为 `gen` 的模块构建的。它包装了 `sys` 和 `proc_lib` 模块，处理了前面章节中讨论过的许多并发和分布式编程相关的棘手问题与特殊情况。你可以在 `stdlib` application 的源代码目录中找到它的源代码。如果你正在实现你自己的 behavior，不想被一些棘手问题绊倒，也许你会想使用 `gen` 而不是自己从头来。不过请注意，它没有配套文档，而且未来可能会在不对外通知的情况下被修改。

创建 behavior 时的要求

创建自己的 behavior 的步骤很简单，需要将代码分解为通用模块和专用模块，并定义相应的回调函数及其返回值。做的时候，需要遵循以下这些简单的规则：

- 通用模块的名称必须与 behavior 名称相同。
- 需要在 behavior 模块中列出回调函数。
- 在回调模块中，包含 `-behavior(BehaviorName).` 指令。

behavior 代码的通用模块编译好后，任何使用该 behavior 的回调模块如果遗漏了回调，编译时都会被警告。

一个处理 TCP 流的例子

让我们来看一个实现自己的 behavior 的例子，重点关注与我们实现 behavior 相关的代码。我们省略了与示例不相关的函数，在代码中将其标记为 ...（省略号）。如果你想看完整的模块，可以在本书的代码库中找到。但是，如果你只想弄明白实现自己的 behavior 时的细节，则无须查看完整示例。

我们的例子是一个包装器，它封装了与 TCP 流相关的活动，包括连接、配置和错误处理等，只暴露出正在接收的数据流。每次收到一个套接字请求后，behavior 会产生一个新的进程，只要套接字处于打开状态，该进程就会继续存活。behavior 会接收数据包，并在到达时将其转发到回调模块。套接字可以由回调模块通过回调函数的返回值关闭，或者当 TCP 客户端关闭连接时间接地关闭。

回调模块中的回调函数由一个初始化函数（在打开套接字时调用一次）、一个数据处理函数（每收到一块数据就会调用一次），以及一个在套接字关闭时调用的终止函数组成：

```
-module(tcp_print).  
-export([init_request/0, get_request/2, stop_request/2]).  
-behavior(tcp_wrapper).
```

◀ 257

```

init_request() ->
    io:format("Receiving Data~n."),
    {ok, []}.
get_request(Data, Buffer) ->
    io:format("."),
    {ok, [Data|Buffer]}.
stop_request(_Reason, Buffer) ->
    io:format("~n"),
    io:format(lists:reverse(Buffer)),
    io:format("~n").

```

回调函数 `init_request/0` 返回 `{ok, LoopData}`。函数 `get_request/2` 接收绑定到变量 `Data` 的 TCP 数据包以及 `LoopData`，并返回 `{ok, NewLoopData}` 或 `{stop, Reason, NewLoopData}`。在这个例子中，`LoopData` 是绑定到变量 `Buffer` 的用于接收 TCP 数据包的缓冲区。在关闭套接字后，`stop_request/2` 被调用并收到终止原因 `Reason` 和 `LoopData`，并且必须返回原子 `ok`。

请注意，我们在代码中包含了 `-behavior(tcp_wrapper)` 指令，其指向 `tcp_wrapper` 模块，也正是 `behavior` 实现的地方。

当启动 `tcp_wrapper` `behavior` 时，我们传给它回调模块 `Mod` 和端口号 `Port`。我们产生了一个进程，初始化 `behavior` 状态，打开一个套接字监听，并最终进入 `accept/4` 函数。针对每个并发流，我们都会在监听的套接字上接受连接，并产生一个新进程从 `init_request/2` 函数开始执行，进而通过回调模块处理流。在 `accept` 调用中，指定了一个超时以防止无限阻塞，所以我们可以每秒将控制返回到主循环（在此示例中未展示），以确保可以处理系统消息和来自父进程的 `EXIT` 信号。我们还导出了 `cast/3` 函数，它允许创建一个连接并向服务器异步发送请求：^{注1}

```

-module(tcp_wrapper).
-export([start_link/2, cast/3]).
-export([init/3, system_continue/3, system_terminate/4, init_request/2]).

-callback init_request() -> {'ok', Reply :: term()}.
-callback get_request(Data :: term(),
                      LoopData :: term()) ->
    {'ok', Reply :: term()} |
    {'stop', Reason :: atom(), LoopData :: term()}.
-callback stop_request(Reason :: term(), LoopData :: term()) -> term().

```

注1 除了这种基于超时的方案，另一种替代方法是使用 `prim_inet:async_accept/2` 函数，当接受新的连接时，该函数会向调用进程发送消息，但该函数本是作为 Erlang / OTP 内部使用的函数，因此属于未文档化和公开支持的 API 函数。


```

start_link(Mod, Port) ->
    proc_lib:start_link(?MODULE, init, [Mod, Port, self()]).

cast(Host, Port, Data) ->
    {ok, Socket} = gen_tcp:connect(Host, Port, [binary, {active, false},
                                              {reuseaddr, true}]),
    send(Socket, Data),
    ok = gen_tcp:close(Socket).

send(Socket, <<Chunk:1/binary,Rest/binary>>) ->
    gen_tcp:send(Socket, [Chunk]),
    send(Socket, Rest);
send(Socket, <<Rest/binary>>) ->
    gen_tcp:send(Socket, Rest).

init(Mod, Port, Parent) ->
    {ok, Listener} = gen_tcp:listen(Port, [{active, false}]),
    proc_lib:init_ack({ok, self()}),
    loop(Mod, Listener, Parent, sys:debug_options([])).

loop(Mod, Listener, Parent, Debug) ->
    receive
        {system,From,Msg} ->
            sys:handle_system_msg(Msg, From, Parent,
                                  ?MODULE, Debug, {Listener, Mod});
        {'EXIT', Parent, Reason} ->
            terminate(Reason, Listener, Debug);
        {'EXIT', Child, _Reason} ->
            NewDebug = sys:handle_debug(Debug, fun debug/3,
                                         stop_request, Child),
            loop(Mod, Listener, Parent, NewDebug)
    after 0 ->
        accept(Mod, Listener, Parent, Debug)
    end.

accept(Mod, Listener, Parent, Debug) ->
    case gen_tcp:accept(Listener, 1000) of
        {ok, Socket} ->
            Pid = proc_lib:spawn_link(?MODULE, init_request, [Mod, Socket]),
            gen_tcp:controlling_process(Socket, Pid),
            NewDebug = sys:handle_debug(Debug, fun debug/3, init_request, Pid),
            loop(Mod, Listener, Parent, NewDebug);
        {error, timeout} ->
            loop(Mod, Listener, Parent, Debug);
    end

```

```

{error, Reason} ->
    NewDebug = sys:handle_debug(Debug, fun debug/3, error, Reason),
    terminate(Reason, Listener, NewDebug)
end.

system_continue(Parent, Debug, {Listener, Mod}) ->
    loop(Mod, Listener, Parent, Debug).

system_terminate(Reason, _Parent, Debug, {Listener, _Mod}) ->
    terminate(Reason, Listener, Debug).

terminate(Reason, Listener, Debug) ->
    sys:handle_debug(Debug, fun debug/3, terminating, Reason),
    gen_tcp:close(Listener),
    exit(Reason).

debug(Dev, Event, Data) ->
    io:format(Dev, "Listener ~w:~w~n", [Event,Data]).

init_request(Mod, Socket) ->
    {ok, LoopData} = Mod:init_request(),
    get_request(Mod, Socket, LoopData).

get_request(Mod, Socket, LoopData) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Data} ->
            case Mod:get_request(Data, LoopData) of
                {ok, NewLoopData} ->
                    get_request(Mod, Socket, NewLoopData);
                {stop, Reason, NewLoopData} ->
                    gen_tcp:close(Socket),
                    stop_request(Mod, Reason, NewLoopData)
            end;
        {error, Reason} ->
            stop_request(Mod, Reason, LoopData)
    end.

stop_request(Mod, Reason, LoopData) ->
    Mod:stop_request(Reason, LoopData).

```

260 处理 TCP 流的通用代码很简单。其进程循环首先初始化流状态，并且接收数据包，然后当回调模块返回一个 stop 元组，或者当 TCP 客户端决定关闭连接端时终止。在初始化、接收数据包和终止时，会调用 Mod 回调模块中相应的回调函数。

我们的 behavior 最引人注目的一个部分——也许是与调用回调函数一样重要的一个

部分——是回调规范，它列出了需要在回调模块中导出的回调函数，描述时基于的是 Erlang 类型和函数规范中的指令。事实上，最终这些回调规范会映射为 `behavior_info(callbacks)` 函数，该函数返回形如 `{Function, Arity}` 的列表。因此你甚至可以完全不写回调规范，直接在通用 `behavior` 模块中实现并导出 `behavior_info/1` 函数（这正是 R15B 之前旧版本 Erlang/OTP 发行包中实现 `behavior` 的方式）。比较回调规范与 `tcp_print` 模块中的回调函数看看，它们匹配吗？

```
-module(tcp_wrapper).
...
-export([behavior_info/1]).

behavior_info(callbacks) ->
    [{init_request, 0}, {get_request, 2}, {stop_request, 2}].
...
```

与 `behavior_info/1` 函数相比，使用回调规范的优点是，*dialyzer* 工具会检测回调模块是否遵循了回调规范，能警告你回调函数未定义。*dialyzer* 默认启用 `behavior` 回调警告。记得在编译回调模块之前先编译通用的 `behavior` 模块，并将其置于代码搜索路径中，否则你会得到一个 `behavior` 未定义的警告。

总结

在本章中，我们详细介绍了实现特殊进程的方法，以及如何使其符合 OTP 标准，并将其纳入 OTP 监督树的一部分。不仅如此，我们还做了更进一步的设计，将代码拆分为通用模块和专用模块，制作成了完整的 `behavior`——不仅有回调模块，还有 `behavior` 指令以及相关的编译器警告。

在启动和休眠特殊进程时，必须使用表 10-1 中列出的 `proc_lib` 模块中的功能，而不是标准的 BIF。

表 10-1: 通过 `proc_lib` 模块启动特殊进程

函数调用	回调函数或动作
<code>proc_lib:spawn_link/1,2,3,4</code>	无
<code>proc_lib:spawn_opt/2,3,4,5</code>	无
<code>proc_lib:start/3,4,5</code>	<code>proc_lib:init_ack(Parent, Reply)</code> , <code>proc_lib:init_ack(Reply)</code>
<code>proc_lib:start_link/3,4,5</code>	<code>proc_lib:init_ack(Parent, Reply)</code> , <code>proc_lib:init_ack(Reply)</code>
<code>proc_lib:hibernate/3</code>	无

表 10-2 中列出的系统消息调用及它们对应的回调都需要由你的进程负责处理，你可以选择直接向发来请求的进程做出回应，也可以使用 `sys` 模块来完成。

表 10-2: 系统请求与消息

消息	回调函数或动作
<code>{system, From, Request}</code>	<code>Mod:system_continue(Parent, Debug, LoopData), Mod:system_terminate(Reason, Parent, Debug, LoopData)</code>
<code>{'EXIT', Parent, Reason}</code>	<code>exit(Reason)</code>
<code>{get_modules, From}</code>	<code>From ! {modules, ModuleList}</code>

可以在 `sys` 和 `proc_lib` 模块各自的手册页中阅读更多信息。在 OTP 设计原则用户指南的“`sys` 与 `proc_lib`”部分有一个示例介绍了特殊进程和用户自定义 `behavior` 方面的知识。最后，在定义你自己的回调时，涉及的类型和函数规范等信息可以在 Erlang 参考手册和用户指南中找到。

如果你喜欢编写代码，建议你从本书的代码库下载互斥体示例，并构造一些有可能在并发应用程序中出现的特殊情况进行测试。在你的客户端函数中，当请求互斥锁时，为保证你发送的回应是有效的，可以在其中增加引用，并且还可以带上可选的超时值。除此之外，考虑到在临界区内执行代码时可能出现异常终止的情况，你还需要考虑对互斥锁做监视。

接下来是什么

特殊进程与用户自定义 `behavior` 是重要的基础手段，借助它们可以构建出已有的 `behavior` 以及创建新型的 `behavior`，让我们将进程纳入监督树，并打包在一个 `application` 中。在接下来的一章中，我们将把一系列 `application` 组合在一个发行包里，并且你会看到我们如何配置、启动和停止一个完整 Erlang 节点。

系统原则与发行包制作

经过前面的介绍，我们已经知道如何实现和使用已有的 OTP behavior，将它们组织成带有特殊进程的监督树，并打包到 application 里的方法，接下来我们将介绍如何将这些 application 组合成一个 Erlang 节点，使其能够作为一个单元来启动。在许多编程语言中，打包 (packaging) 是交由操作系统负责的。但在 Erlang 中，可通过直接在 OTP 中创建所谓的 release (发行包) 来完成，根据情况不同，单个系统可能对应多个不同的发行包。每个 Erlang 节点各自运行着一个发行包，既可以是单机的方式，也可以是分布式的方式。制作发行包需遵循一定的标准，它可使你的系统按一种通用的结构组织而成，该结构不仅具备目标独立性，而且能够使用独立于底层操作系统的工具进行管理和升级。因此，虽然 Erlang 的发行包制作过程可能初看起来较复杂，但事实上与创建非 Erlang 包相比，它是比较容易的。如果我们回顾 Erlang 的层次化封装结构，不难想起，我们是从 function 开始，然后到 module，再到 application 层层递进的。而一个 Erlang 节点实际上又更进一步，是由若干个松散耦合的 application 打包为一个发行包后构成的。

你可能尚未意识到，当你在计算机中安装 Erlang 时，其实安装的是一个标准发行包。标准发行包与你自己创建的发行包之间的不同之处在于所加载和启动的 application 不同，以及它们的配置参数不同。底层的 Erlang 运行时系统实际上并不会区别对待用户的 application 和 Erlang / OTP 自带的 application。发行包都具有相同的目录结构，自己的一份虚拟机副本，并以相仿的方式管理发行包和配置文件。正因为如此，你用 erl 命令启动的那个 Erlang 发行包，其制作所使用的基础工具、目录结构和系统原则，与你制作自己的发行包时所用的完全相同。

在本章中，我们会引导你完成构建发行包所需的步骤，并解释各个步骤之间是如何相互衔接的。我们会介绍不同发行包类型，一类是简单并具备交互功能的目标系统——这类系统的模块加载支持更灵活，并允许你在运行过程中方便地启动 application；另一类则

是嵌入式目标系统——在这类系统中，模块加载和 application 启动操作都受严格的版本控制。为了创建目标系统，我们会介绍 `systools`，它是一个 Erlang 库，当你想把发行包制作过程与现有的工具链或构建过程集成时会用到它；我们还会介绍 `rebar3` 的使用，它适合无历史累赘的全新项目（greenfield projects）或依赖项管理变得复杂的情形。

系统原则

一个 Erlang 发行包被定义为一个独立的节点，其中包含：

- 一些 OTP application，它们是针对项目的某些功能而编写（或重用而来）的，通常包含的是系统的业务逻辑。application 可以是专有的（proprietary），可以是开源的，或此二者的组合。
- 来自标准分发所自带的一些 OTP application，上述 application 依赖于这些 OTP application。
- 一些配置文件、引导（boot）文件，以及一个启动脚本。
- Erlang 运行时系统本身，其中包括一份虚拟机的副本。

有一些工具可以帮助你创建和打包独立节点，但是在介绍它们之前，我们将详细介绍所有组件，并指导你手动完成构建。这将帮助你更好地了解发行包的结构和工作方式，以及你能够控制的各种选项。

启动 Erlang 节点的最简单方法是使用 `erl` 命令。要想进一步启动你的程序，可以直接在 Erlang shell 里输入模块和函数名称，或是在一开始将 `-s` 标志传递给 `erl`：

```
$ erl -s module function arg1 arg2 ...
```

`function` 和参数可以省略。如果仅指明 `module`，则该命令将导致调用 `module:start()`。如果指明了 `module` 和 `function`，该命令将导致调用 `module:function()`。我们把用这种方法启动的节点称为基本目标系统（basic target system），你可以写一个 UNIX shell 脚本来初始化状态并调用 `erl -s` 命令。但这种方法只能用于开发阶段，用于概念验证或临时探查。不建议在生产环境中使用基本目标系统，因为这将使你失去许多 OTP 原本能为你带来的好处。你其实有更好的选择。

265



除非只是做概念验证或快速探查，否则不要运行基本目标系统。通过调用 `erl -s myprojectsup -noshell` 方式启动你的程序，将使你失去所有由 OTP application 启动、监督和升级过程所带来的好处。改为使用引导（boot）文件并以嵌入式目标系统（embedded target system）方式运行系统，将使你重新获得所有的好处。

启动节点的另一种方式是简单目标系统 (simple target system)。它采用了 *sasl* application 所附带的引导脚本和工具，能享受运行时执行软件升级的便捷功能。为了搞明白简单目标系统是如何工作的，我们首先检查一下你安装好的 Erlang，并调查它的目录结构以及与之相关的所有文件和脚本。在生成发行包时，你需要使用诸如 *systools*、*reltool* 或 *relx* 之类的工具自己生成其中的一些文件，但是你也可以从代码仓库或目标环境安装好的 Erlang 中复制出这些文件来使用。

首先找到顶层目录，通常我们称之为 *Erlang* 根目录 (Erlang root directory)。这是你（或你使用的脚本）安装 Erlang 的位置。如果你不知道该位置，请启动 Erlang 节点并调用 `code:root_dir().`：

```
$ erl
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V7.2 (abort with ^G)
1> code:root_dir().
"/usr/local/lib/erlang"
2> q().
ok
$ cd /usr/local/lib/erlang
$ ls
Install    erts-6.4      erts-7.1    misc
bin        erts-6.3      erts-7.2    releases
erts-6.2   erts-7.0      lib         usr
```

目录中的内容事实上正是创建发行包时所获得的输出。但是具体内容根据你安装 Erlang 的方式不同（以及来源不同）、多年来完成的升级次数不同以及构建该发行包的人员所做的定制不同而有所不同。但是，有一些基本文件和目录是必需的，会始终存在，并且会在第一次安装时出现。

发行包目录结构

在本小节中，我们将探索制作发行包所涉及的文件。你自己的发行包的目录和文件结构实际上与 Erlang 根目录一样，因此我们会花点时间来研究一下它。Erlang 根目录与你自己的发行包之间存在的差异主要是加载和启动哪些 application，以及这些 application 的版本可能有所不同，另外运行时系统的版本也可能不同。在接下来的几节，我们会创建一个基站控制器 (base station controller) 发行包，其遵循了我们提过的原则，你将会有更深入的理解。

有 4 个目录是每个 OTP 发行包中都必须具备的，如图 11-1 所示。其中 *lib* 目录我们先

◀ 266

前已经查看过，其下包含所有的 application 目录，并且各个目录的命名遵循 application 名称后加对应版本号的形式。先前在第 9 章的“OTP application 的结构”一节，我们已介绍过 application 以及它们对应的目录结构。随着一次次升级，你可能会注意到单个 application 同时存在多个版本的情形，不同版本的 application 可以通过目录名称中的版本号部分相区分。创建发行包时，对于这种多个实例并存的情形，我们所指定的代码搜索路径一般会指向最新版本的 application 的 *ebin* 目录。

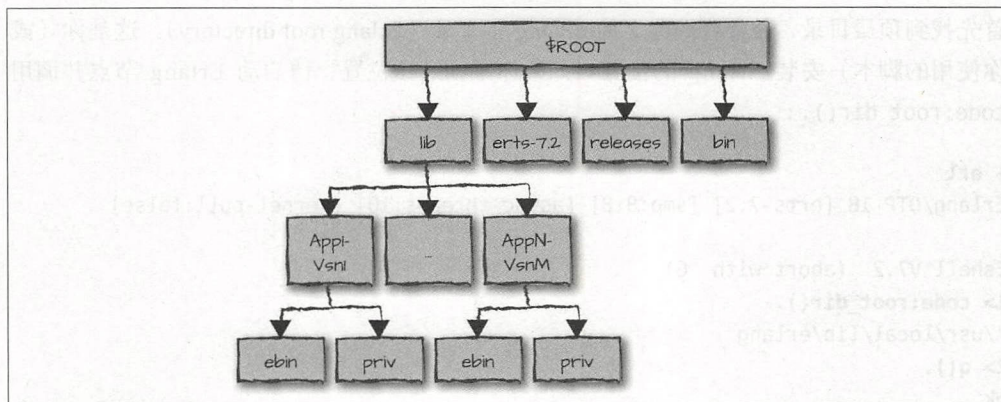


图 11-1: 发行包目录结构

erts 目录内包含的是 Erlang 运行时系统的二进制文件。即使是此目录，如果你曾做过升级安装，也可能会发现该目录的多个实例，以目录名称后附加的 *erts* 不同版本号相区分。最有趣的子目录是 *bin*，其中不仅包含与虚拟机相关的可执行文件和 shell 脚本，还包含各种能够从 shell 调用的工具。环顾其中，你会发现：

erl

一个脚本或程序（具体是哪种取决于目标环境），用于启动运行时系统并提供一个交互式的 shell。

erlexec

一个二进制可执行文件，被 *erl* 脚本调用。

erlc

Erlang 相关的各种编译器。根据你正在编译的文件的扩展名不同会选择不同的编译器。

267 *epmd*

Erlang 端口映射器（port mapper）守护进程。在分布式 Erlang 环境下，它充当了名称服务器的角色，将 Erlang 节点映射到 IP 地址和端口号。

escript

允许你像执行脚本一样直接执行简短的 Erlang 程序，而不必编译它们。

start

用于在 UNIX 环境中以嵌入式目标系统模式启动 Erlang 系统。此类发行包会作为一个守护进程运行，没有 shell 窗口。我们会在本章后面“打包发行包”一节中介绍嵌入式目标系统。

run_erl

此二进制文件会被 *start* 所调用，用于启动基于 UNIX 的嵌入式目标系统，其 I/O 是通过管道流式传输的。

to_erl

在嵌入式目标系统中连接到由 *run_erl* 启动的节点的 Erlang I/O 流。

werl

在 Windows 环境中启动运行时系统，与控制台分属不同窗口。

start_erl

属于启动嵌入式目标系统时的命令链中的一环，如果是在 UNIX 系统中，作用是设置启动文件和配置文件等；而在 Windows 环境中，则与前面描述的 *start* 命令类似。

erlsrv

与 *run_erl* 类似，但其是针对 Windows 环境的，允许 Erlang 在无须用户登录的情况下启动。

heart

监视 Erlang 运行时系统的心跳，并在收不到心跳时调用某个脚本。

dialyzer

一个用于 beam 文件和 Erlang 源代码文件的静态分析工具。它能够发现各类问题，包括类型差异、死代码或不可达代码等。每个人都应该把 *dialyzer* 纳入构建过程的一部分。

typer

在 Erlang 程序中根据变量的使用情况推断变量的类型。它为源代码增加了类型规范，并将作为输入数据提供给 *dialyzer*。

在创建和启动 Erlang 发行包时，程序员会用到 bin 目录中列出的一部分可执行文件。此处列出的是最重要的，与我们将在本章后面详细介绍的内容紧密相关。但是这个列表并

不完整，因为完整的内容取决于你正在运行的 Erlang/OTP 版本以及对应的操作系统。

`erts-version/bin` 目录里的内容与 Erlang 根目录的 `bin` 目录里的内容很相似。事实上，Erlang 根目录中的 `bin` 目录下的脚本和可执行文件是从带版本号的 `erts-version/bin` 目录下复制的，甚至前者中有文件是以链接的形式指向后者中的。但是 Erlang 根目录中的 `bin` 目录依然是必需的，因为你可能安装过同一个发行包的多个版本，并且同时运行着它们。当你键入 `erl` 时指向的虽然是根 `bin` 目录中的脚本，但其中的环境变量会将其真正的目标重定向到你正在使用的 `erts-version/bin` 下。我们来看看 `erl` 脚本的内容。此处以一台运行着 OS X Yosemite 操作系统的 Mac 为例，它所用的 18.2 版发行包的 `erl` 脚本内容如下所示。

```
#!/bin/sh
#
# %CopyrightBegin%
#
# Copyright Ericsson AB 1996-2012. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# %CopyrightEnd%
#
ROOTDIR="/usr/local/lib/erlang"
BINDIR=$ROOTDIR/erts-7.2/bin
EMU=beam
PROGNAME=`echo $0 | sed 's/.*\\///'`
export EMU
export ROOTDIR
export BINDIR
export PROGNAME
exec "$BINDIR/erlexec" ${1+"$@"}
```

`ROOTDIR` 和 `BINDIR` 以及其他环境变量是在安装或升级 Erlang 时设置的。请注意，其中 `BINDIR` 指向了我们在本节开头曾探索过的 `$ROOTDIR/erts-7.2/bin` 目录，并在最后执行

了 *erlexec*。请查看 *erts-7.2/bin* 目录中的 *erl.src*，它是用于创建 *erl* 脚本的源文件。类似的，针对 *start_erl* 和 *start* 也存在类似的源文件。我们在本章后面“打包发行包”一节中会介绍 *.src* 文件。

如果进入位于 Erlang 根目录下的 *releases* 目录，你会发现其中针对你安装过的每一个版本的发行包都有一个对应的目录。它们每一个应该都各对应一个 *erts* 目录，因为大多数新版本发行包都带有新版本的运行时系统。检查 *start_erl.data* 文件中的内容，你将看到两个数字，第一个表示当前安装的 Erlang 中使用的虚拟机版本，第二个表示当前使用的 OTP 发行包的目录：

```
$ cd releases
$ ls
17      18 RELEASES      RELEASES.src    start_erl.data
$ cat start_erl.data
7.2 18
$ ls 18
OTP_VERSION                start.boot        start_sasl.boot
installed_application_versions start.script        start_sasl.rel
no_dot_erlang.boot         start_all_example.rel start_sasl.script
no_dot_erlang.rel           start_clean.boot
no_dot_erlang.script        start_clean.rel
```

如果查看 *start_erl.data* 中所指的目录的内容（此示例中我们选择了版本 18），则可以找到一些包含以 *rel*、*script* 和 *boot* 为扩展名的文件。后缀为 *rel* 的文件中列出的是其所属的发行包中的 *application* 和运行时系统的版本清单。后缀为 *boot* 的文件则是 *.script* 文件的二进制版，其中包含一系列加载和启动 *application* 的命令，会在系统刚启动时执行。你不妨进入最新版发行包所对应的子目录，然后查看任意的 *.rel* 和 *.script* 文件，以了解它们可能执行的操作。在接下来的小节里，我们将自己创建这些脚本。

你可以通过设置 *sasl application* 的配置变量 *releases_dir* 或更改 OS 环境变量 *RELDIR* 来改变 *releases* 目录的默认位置。Erlang 运行时系统必须具有对此目录的写入权限才能完成升级操作，因为升级时会更新 *RELEASES* 文件。

发行包资源文件

你的项目中全部的 OTP *application*（包括复制自标准发行包的、专有的和开源的等）捆绑在一起，连带它们的版本号构成了一个发行包规范（*release specification*）。此规范中还带有发行包版本及其名称，以及运行时系统的版本等。构建系统使用这些信息进行完整性检查、创建引导文件与目标目录结构等。

最小的（和默认的）发行包只包含 *kernel* 和 *stdlib* 两个 *application*，但是大多数发行包

270 还包含并启动 *sasl*，因为其提供了软件升级所需的全套工具。在刚开始学习创建第一个发行包时，你大概不会考虑到升级问题，但之后某个时候你很可能会有此需求。以源代码方式安装 Erlang 时，你可以选择是否包含 *sasl*（默认是包含的），但如果你使用的是第三方制作好的二进制版本完成的安装，则此选项是别人为你选择的。

让我们更进一步看看 *rel* 文件。如果从 Erlang 根目录进入 *releases* 目录，并从那里进入任何一个子目录，你至少会找到一个以 *rel* 为后缀的文件。作为例子，我们选择了 *releases/18/start_sasl.rel* 文件，并删除了其中的注释：

```
{release, {"Erlang/OTP", "18"}, {erts, "7.2"},
  [{kernel, "4.1.1"},
   {stdlib, "2.7"},
   {sasl, "2.6.1"}]}.
```

正如我们所看到的，这个发行包将运行的模拟器版本是 7.2，启动的 *kernel* 版本是 4.1.1，*stdlib* 的版本是 2.7，*sasl* 的版本是 2.6.1。该发行包的名称是“Erlang/OTP”，其版本是 18。还可以举出很多这样的 *rel* 文件的例子，其中的版本信息，以及各自目录下相应的引导文件和脚本文件实际上描述了系统是如何被组织到一起的。

你会用到标准发行包中的哪些 application

在我们的基站发行包文件中，为了简单起见，只包含了 *bsc* application。然而真实的生产系统通常还包括一些用于监视、日志记录和调试的 application，它们并不会影响到你的 application，但却为你提供了对线上系统进行故障排除时所需的洞察力和可视性。我们已经在 *sasl* application 中见识了一种基本的日志和警报形式。而 *os_mon* application 提供的则是检查底层操作系统的能力，包括磁盘和内存监视以及 CPU 负载和利用率信息等。

runtime_tools application 经常被忽略和省略。它包括 *dbg* 调试器和 *system_information* 模块，以及其他用于实时分析虚拟机所需的工具。你永远不知道这些工具和它们带来的可见性何时会派上用场（特别是 *dbg*），所以我建议你包括它们。

我们创建一个名为 *basestation.rel* 的发行包文件，用于基站控制器例子。发行包的名称是 *basestation*，我们给它的版本是 1.0。除了标准发行包自带的 application，它还将包含版本为 1.0 的 *bsc*。该文件相当简单，与前面例子中的内容差别很小：

```
{release,
  {"basestation", "1.0"},
  {erts, "7.2"},
  [{kernel, "4.1.1"},
```



```
{stdlib, "2.7"},
{sasl, "2.6.1"},
{bsc, "1.0"}}}.
```

资源文件按照惯例命名为 *ReleaseName.rel*。这个约定并非强制的，但是这样做对那些支持和维护代码的人更友好。该资源文件中包含一个由四个元素组成的元组：第一个元素为 *release* 原子，第二个元素是格式为 *{ReleaseName, RelVersion}* 的元组，第三个元素是格式为 *{erts, ErtsVersion}* 的元组，第四个元素是由包含 *application* 及其版本信息所组成的元组列表。到目前为止，我们看到的 *application* 元组格式都类似 *{Application, AppVersion}*，但事实上如下所示，还存在其他格式：

```
{release,
 {ReleaseName, RelVersion},
 {erts, ErtsVersion},
 [{Application, AppVersion},
 {Application, AppVersion, Type},
 {Application, AppVersion, IncludedAppList},
 {Application, AppVersion, Type, IncludedAppList}]
}.
```

元组中各个元素的版本字段都是字符串。在你的 *application* 元组中，也可以添加一个 *Type*。我们在第 9 章的“*application* 类型与终止策略”一节曾介绍过的类型你在此都可以使用，并且还能填写为 *load* 或 *none*：

load

加载 *application* 但不启动它。

none

加载 *application* 中的模块，但不加载 *application* 本身。

permanent

顶级监督者终止时关闭整个节点。当 *application* 终止时，所有其他 *application* 都会随之终止。在没有指定重启类型 (*restart type*) 的情形下，这是默认行为。

transient

当顶级监督者终止时，如果其原因为非 *normal* 则关闭整个节点。这仅适用于不启动自己的监督树的库 *application*，因为顶级监督者总是会以非 *normal* 原因，即 *shutdown* 终止，从而产生与 *permanent application* 相同的结果。

temporary

在 SASL 日志中做记录，但不会影响发行包中的其他 *application*。

272 最后，你可以在 `IncludedAppList` 中指定内含型 `application` 的列表。该列表必须是 `application` 的 `app` 文件中指定的 `application` 的子集。

发行包与 application 版本

每一个 OTP 版本号都对应了一组在 `rel` 文件中列出的 `application` 版本号，以及一个模拟器版本号，它们经过测试能够配合运转。但这并不意味着你不能改变其中 `application` 与模拟器的版本搭配。当然，当你做出改变后你得自己去测试。好在 OTP 发行包的测试用例是其源代码库的一部分，因此你可以在开发过程中自由地去执行这些测试用例。每个 `application` 版本号都对应了一组模块版本号和资源版本号，列在 `app` 文件中或包含在 `priv` 目录中。

从 OTP 17 开始，`application` 版本和 OTP 版本采用了一致的编号方案。它们都由三个整数组成，格式为 `<Major>.<Minor>.<Patch>`。其中主（Major）版本号包含的是实质性的，可能非向后兼容的更改。而次（Minor）版本号则是在添加了新功能时递增，至于补丁（Patch）版本号则是随错误修复而递增的。如果一个发行包，其主版本号升高，那么会将次版本号和补丁版本号重置为 0；但如果只是增加了次版本号，则只需将补丁版本号重置为 0 即可。末尾的 0 通常会从版本号中移除，因此版本 17.1.0 等同于版本 17.1。

主版本号升高后的新发行包会包含所有先前的次版本号和补丁版本号中所做的修改。排除可能已被删除的功能和向后不兼容的变更部分，你可以假定较高版本包含了低版本中所有的错误修复和功能增强。

版本号可以包含超过三个部分。这使你能够指明所创建的发行包所属的分支，以便能针对旧发行包发布兼容的修补程序。对于你可以使用多少分支版本号并无限制。例如，`application` 或发行包 17.1.3.1 中的修复程序不能保证会包含在 17.2 中，因为 17.2 可能在 17.1.3.1 之前发布。预发行版（也称为候选发行版）将具有后缀，例如，17-rc1。

如果你不确定当前使用的是哪个版本的 OTP 发行包，可以使用 `erlang:system_info(otp_release)` BIF 获取。在当前运行的发行包的 `releases` 目录中，你会找到包含 OTP 版本号的 `OTP_VERSION` 文件。但你只能在你的开发环境中找到这个文件。如果在你的目标安装中寻找它，除非你自己把它放在那里，否则是不会找到的。

273 创建发行包

明确了打算在发行包中包含的内容后，现在只需几个简单的步骤就能创建出它，如

图 11-2 所示。

1. 首先创建一个二进制 boot（引导）文件，其中包含加载模块和启动 application 所需的命令。
2. boot 文件准备好后，创建一个目录结构，其中包含所有的 application 目录、release 目录以及（如果需要的话）模拟器。这个软件包是目标独立的（target independent），但可能是与操作系统和硬件相关的。你的目录结构必须遵循本章前面“发布包目录结构”小节所介绍过的，确保其与你创建的 boot 文件相互兼容。
3. 创建一个启动脚本，其中设定了配置信息、系统限制、代码搜索路径和其他系统特定的环境变量等，并包含一个指向引导文件的指针。该脚本将基于你曾在模拟器的 bin 目录中看到过的 .src 文件建立。此外，该脚本将依赖于你创建的目录结构以及你希望目标系统以何种方式运作等。
4. 启动脚本创建好后，需要创建一个针对你的目标环境的部署包。具体形式可以是一个 tar 文件、Debian 或 Solaris 软件包、容器镜像或任何其他你认为便于部署和配置的形式。

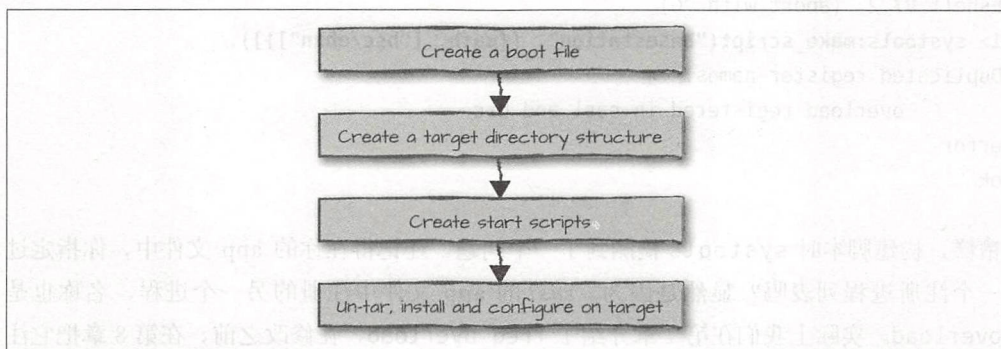


图 11-2: 创建一个 OTP 发行包

在我们的例子中，为简单起见，我们选择创建和部署一个 tar 文件，这需要使用 `systools` 库，该库来自 OTP 分发中的 SASL application。典型的目标目录结构中包含 `rel` 文件中列出的所有 application，并且大多数情况下还会包含 Erlang 运行时系统。一旦我们创建好了 tar 文件，接下来就解压缩它，并更改其中的脚本、配置文件以及其他与目标环境有关的环境变量等，然后才做打包。这一步可以手动完成，也可以作为自动构建过程的一部分。你可以在你的计算机上直接来做这些，也可以在你的目标环境中去做。具体怎么做取决于开发需求、目标环境以及你选择的工具等。从来没有也永远不会“通吃”的方案。

创建 boot 文件

我们开始着手创建 boot（引导）文件吧。为此，我们需要用到 `systools:make_script/2` 库函数。这个函数会创建一个二进制 boot 文件，start 脚本会用它来启动 Erlang 和你的系统。要想让 `start_script/2` 函数正常工作，我们需要复制 *bsc* application 示例，确保它遵循第 9 章“OTP application 的结构”部分所介绍的目录结构。该结构可在本章的 GitHub 存储库目录中找到。如果你下载并在计算机上重演该示例，不要忘记编译 Erlang 并将文件放置到 *ebin* 目录中。

脚本首先查找 *basestation.rel* 文件中指定的那些 application 的版本。这一过程中会用到代码搜索路径，并且如果你设置了环境变量 `{path, PathList}` 则也会被用到。在我们的例子中，假设在与 *bsc* 同目录中使用 `erl -pa bsc/ebin`，或者是使用 `[{path, ["bsc/ebin"]}]` 选项启动了 Erlang。请记住，`PathList` 是列表的列表，因此即使只有一个目录，也必须在列表中定义该目录：`[Dir]`。让我们试试看：

```
$ erl -pa bsc/ebin/  
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> systools:make_script("basestation", [{path, ["bsc/ebin"]}]).
```

```
Duplicated register names:
```

```
    overload registered in sasl and bsc
```

```
error
```

```
ok
```

糟糕，构建脚本时 `systools` 检测到了一个问题。还记得在你的 *app* 文件中，你指定过一个注册进程列表吗？显然是因为，*sasl* 的 *app* 文件中注册的另一个进程，名称也是 *overload*。实际上我们在第 7 章介绍了 *freq_overload*，在修改之前，在第 8 章把它注册为了名字 *overload*。所以问题出在当初创建第一个 *app* 文件时，我们使用了一个错误的名称。

如果你在笔记本电脑上运行该脚本，则可能会收到错误提示，脚本无法找到某个版本的 *app* 文件，事实上，随便更改一下 *basestation.rel* 中的任何一个版本号，都能轻松重现该错误。这正是为什么版本控制很重要的原因。你需要确切地知道你在生产中运行的模块、application 和发行包的版本，因为你的系统可能会运行数年并且可能由其他人管理。如果你被叫来处理别人引发的问题，至少你应该知道你要处理的是什麼版本的问题。

在创建 boot 文件时，将做如下检查：

- 检查在 *rel* 文件中定义的所有 application 的一致性和依赖性。是否所有的

application 都存在，并且没有循环依赖关系？确保 app 文件中定义的版本与 rel 文件中指定的版本相匹配。

- 确保 *kernel* 和 *stdlib* 是发行包的一部分，并且它们的类型为 *permanent*。如果 *sasl* 不是发行包的一部分，则会引发警告，但脚本和引导文件还是可以正常执行的。在创建 boot 文件时，你可以通过将 *no_warn_sasl* 作为选项传入来关闭这些警告。
- 检测各个 application 的 app 文件中定义的注册进程名称中可能存在的冲突，确保没有任何两个进程使用了相同注册名称。
- 确保 app 文件中定义的所有模块在 *ebin* 目录中都有相应的 beam 文件存在。这样做的同时，还要检测可能存在的模块名称冲突，主要是相同的模块（或模块名称）包含在多个 application 中。如果你想检查 beam 文件是否对应的是最新版本的源代码文件，可以在选项中包含 *src_tests*。

检查发行包时发现，注册进程名称冲突其实是因为我们的 app 文件中有错。于是我们将 *bsc.app* 文件中的 *overload* 更改为 *freq_overload* 就修复了此问题。

如下例所示，查看输出到目录中的内容时，我们发现了两个新文件，*basestation.script* 和 *basestation.boot*。在进一步研究它们之前，让我们先使用启动文件来启动此基站发行包看看：

```
1> systools:make_script("basestation", [{path, ["bsc/ebin"]}]).
ok
2> q().
ok
$ ls
basestation.boot      basestation.rel      basestation.script    bsc
$ erl -pa bsc/ebin -boot basestation
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
=PROGRESS REPORT==== 25-Dec-2015::20:37:46 ===
```

```
supervisor: {local,sasl_safe_sup}
```

```
started: [{pid,<0.35.0>},
```

```
          {id,alarm_handler},
```

```
          {mfargs,{alarm_handler,start_link,[]}},
```

```
          {restart_type,permanent},
```

```
          {shutdown,2000},
```

```
          {child_type,worker}]
```

...<snip>...

=PROGRESS REPORT==== 25-Dec-2015::20:37:46 ===

supervisor: {local,bsc}

```

started: [{pid,<0.43.0>},
          {id,freq_overload},
          {mfargs,{freq_overload,start_link,[]}},
          {restart_type,permanent},
          {shutdown,2000},
          {child_type,worker}]

```

=PROGRESS REPORT==== 25-Dec-2015::20:37:46 ===

supervisor: {local,bsc}

```

started: [{pid,<0.44.0>},
          {id,frequency},
          {mfargs,{frequency,start_link,[]}},
          {restart_type,permanent},
          {shutdown,2000},
          {child_type,worker}]

```

=PROGRESS REPORT==== 25-Dec-2015::20:37:46 ===

supervisor: {local,bsc}

```

started: [{pid,<0.45.0>},
          {id,simple_phone_sup},
          {mfargs,{simple_phone_sup,start_link,[]}},
          {restart_type,permanent},
          {shutdown,2000},
          {child_type,worker}]

```

=PROGRESS REPORT==== 25-Dec-2015::20:37:46 ===

application: bsc

started_at: nonode@nohost

Eshell V7.2 (abort with ^G)

1> observer:start().

ok

由于 *bsc* application 未放置在 *lib* 目录中，因此我们必须使用 *erl* 命令的 *-pa* 指令将 *.app* 和 *.beam* 文件所在的目录添加进代码搜索路径。注意，*sasl* 启动时立即出现了所有进度报告（我们已经在示例中删除了一些）。为了能完全确保监督树已经启动，请启动 *observer* 工具，选择 Applications 选项卡（参见图 11-3），并查看 *bsc* 监督树。

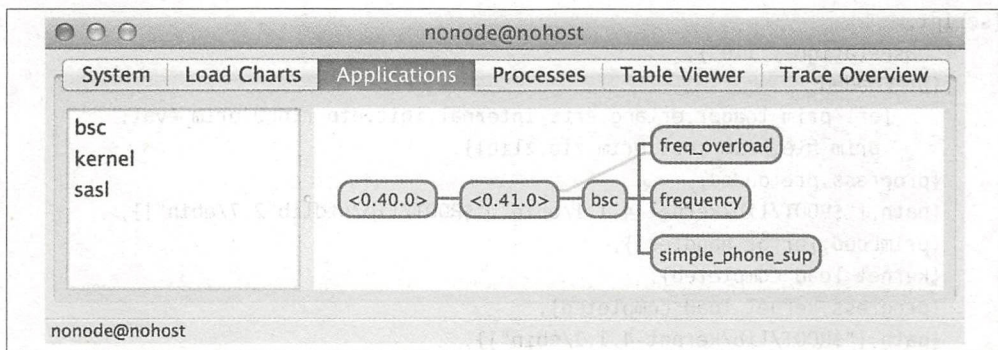


图 11-3: Applications 选项卡

至此我们展示了如何启动符合 OTP 标准的简单目标系统。不少流行的开源项目使用的都是简单目标系统，它比基本目标系统更强大，我们朝着正确的方向又迈出了一步。但我们可以（而且将要）做得更好！在继续探索之前，我们更详细地回顾一下我们生成的文件内容以及可以传递给 `systools:make_script/2` 的参数。

◀ 277

脚本文件

图 11-4 显示了用于构建发行包的几种文件之间的相互关系。*basestation.boot* 是一个二进制文件，其中包含了一系列命令，会被 Erlang 运行时系统执行，发行包要想启动必须用到此文件。与我们曾见过的其他文件不同，此引导文件必须是二进制的，原因是在执行此引导文件的过程中，运行时系统中负责解析和解释文本文件的相关模块会在该引导文件包含的命令的指导下加载。你可以在 *basestation.script* 中找到 *basestation.boot* 文件中的命令的原始文本。令人高兴的是，如果你喜欢自己动手改造，直接编辑这些文件或编写你自创的文件都是可以的。（当你轻松做这些事时，同情一下 1996 年 `make_script/2` 还不存在时，那些使用 OTP R1 的可怜人吧！）

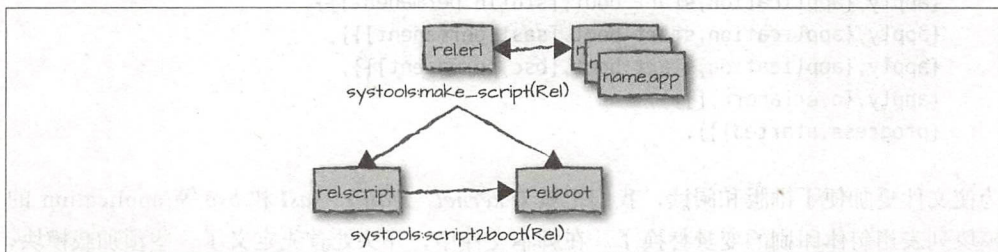


图 11-4: 创建引导文件和发行包文件

看看 `script` 文件的内容吧。它是一个包含了格式为 `{script, {ReleaseName, ReleaseVsn}, Actions}` 的 Erlang 数据项的文件：

```

{script,
  {"basestation","1.0"},
  [{preloaded,
    [erl_prim_loader,erlang,erts_internal,init,otp_ring0,prim_eval,
     prim_file,prim_inet,prim_zip,zlib]],
   {progress,preloaded},
   {path,["$ROOT/lib/kernel-4.1.1/ebin","$ROOT/lib/stdlib-2.7/ebin"]},
   {primLoad,[error_handler]},
   {kernel_load_completed},
   {progress,kernel_load_completed},
   {path,["$ROOT/lib/kernel-4.1.1/ebin"]},
   {primLoad,KernelModuleList}, %%
   {path,["$ROOT/lib/stdlib-2.7/ebin"]},
   {primLoad,StdLibModuleList},
   {path,["$ROOT/lib/sasl-2.6.1/ebin"]},
   {primLoad,SASLModuleList},
   {path,["$ROOT/lib/bsc-1.0/ebin"]},
   {primLoad,BscModuleList},
   {progress,modules_loaded},
   {path,
    ["$ROOT/lib/kernel-4.1.1/ebin","$ROOT/lib/stdlib-2.7/ebin",
     "$ROOT/lib/sasl-2.6.1/ebin","$ROOT/lib/bsc-1.0/ebin"]},
   {kernelProcess,heart,{heart,start,[]}},
   {kernelProcess,error_logger,{error_logger,start_link,[]}},
   {kernelProcess,application_controller,
    {application_controller,start,KernelAppFile}},
   {progress,init_kernel_started},
   {apply,{application,load,StdLibAppFile}},
   {apply,{application,load,SASLAppFile}},
   {apply,{application,load,BscAppFile}},
   {progress,applications_loaded},
   {apply,{application,start_boot,[kernel,permanent]}},
   {apply,{application,start_boot,[stdlib,permanent]}},
   {apply,{application,start_boot,[sasl,permanent]}},
   {apply,{application,start_boot,[bsc,permanent]}},
   {apply,{c,erlangrc,[]}},
   {progress,started}}].

```

为使文件更加便于排版和阅读，我们把属于 *kernel*、*stdlib*、*sasl* 和 *bsc* 等 application 的模块列表用斜体印刷的变量替换了。在脚本文件中，开头处首先定义了一些预加载模块，只有成功加载这些模块后，才能创建进程。让我们逐个学习这些命令。如果你只是想启动并运行系统，那么并不需要深入了解它们的含义，但掌握这一系列启动步骤的知识有助于你深入认识核心系统，并在需要排查系统无法启动（甚至更揪心的无法重新启动）的故障时派上用场：

preLoaded

包含了一个 Erlang 模块列表，在允许任何进程启动之前必须先加载好该列表中的模块。你可以在位于 *lib* 目录里的 *erts application* 中找到其中的各个模块。与本节相关的是 *init* 模块，其中包含了用于解释执行引导文件的代码，以及 *erl_prim_loader* 模块，其中包含了有关如何获取和加载模块的信息。

279

progress

用于报告初始化程序的进度状态。可以随时通过调用函数 *init:get_status/0* 来获取进度状态。该函数返回格式为 {InternalState, ProgressState} 的元组，其中 InternalState 为 starting、started 或 stopping。ProgressState 对应脚本执行时设置的最后一个值。在我们的示例中，与启动过程相关的进度状态只有最后一个 {progress, started}，它使得 InternalState 从 starting 更改为 started。除调试目的外，其他所有阶段值都没有用处。

kernel_load_completed

指明启动任何进程之前所需的所有模块已成功加载。在 *embedded* 模式中此变量被忽略，原因是此场景下所有模块都会在启动系统之前加载。本章后面将更详细地讨论 *embedded* 和 *interactive* 模式。

path

一个路径列表，其中每一个路径均以字符串表示。路径可以是绝对路径，也可以以 *\$ROOT* 环境变量开头。这些路径将（与使用 *-pa*、*-pz* 和 *-path* 作为命令行参数提供的目录路径一同）被添加到代码搜索路径中，并用于加载 *primLoad* 条目中定义的模块。请注意生成的路径中——特别是对应 *bsc application* 的路径中——假定了 *beam* 文件位于目标环境的 *\$ROOT/lib/bsc-1.0/ebin* 下而不是 *bsc/ebin* 下。另外还请注意，启动脚本中的 *application* 版本号被添加到了路径中，假定了其遵循标准的 OTP 目录结构。

primLoad

提供一个模块列表，其中的模块将通过调用 *erl_prim_loader:get_file/1* 函数加载。如果加载模块时出错，则启动脚本将终止并且节点不会启动。当 *beam* 文件丢失、损坏或者编译器对应版本错误，或者代码搜索路径不正确时（例如，你忘记将 *application* 添加到 *lib* 目录或者漏写了目录版本号），都会导致模块加载失败。在本章的多个地方，我们都会介绍如何排查启动错误。

{kernelProcess, Name, {M, F, A}}

通过调用 *apply(M, F, A)* 启动一个核心进程。在我们的文件中，*kernelProcess* 用于三个模块：*heart*、*error_logger* 和 *application_controller*。后两者你已经知

道其用途了。而对于 heart，我们将在本章后面的“heart”小节更详细地进行介绍。核心进程一旦启动完成，就会受到监视，发生任何异常都会导致节点被关闭。

`{apply, {M, F, A}}`

令当前负责初始化系统的进程执行 `apply(M, F, A)` BIF，其中第一个参数是模块，第二个参数是函数，第三个参数是函数的参数列表。如果此功能异常退出，启动过程将中断，并且导致系统终止。被以这种方式调用的函数不允许挂起且必须返回，原因是启动节点的过程是同步式的。倘若 `apply` 不返回，则下一个命令将不会被执行。

现在我们已经对脚本文件中的每一行都有了概括的了解，接下来可以关注其中的一些细节了。

1. 首先预载了 `erts application` 中的所有模块，以及 `kernel application` 中的 `error_handler`。一旦它们加载好，我们便通知脚本解释器 `{kernel_load_completed}` 并发出进度报告。
2. 对于 `release` 文件中列出的所有 `application`，我们将它们的代码路径添加到了代码搜索路径的末尾，并使用 `primLoad` 来加载相应 `application` `app` 文件中列出的所有模块。然后我们发出 `modules_loaded` 进度报告。
3. 从 `heart`、`error_logger` 和 `application_controller`（你已经知道后两个）启动所有的核心进程。然后发出 `init_kernel_started` 进度报告。
4. 调用 `application:load(AppFile)` 加载所有属于此发行包的 `application`。这会加载 `rel` 文件中列出的 4 个 `application`：`kernel`、`stdlib`、`sasl` 和 `bsc`。完成后，发出 `applications_loaded` 进度报告。
5. 至此我们启动了各个核心进程并加载了所有的 `application`，是时候启动它们了。请注意，不是在 `{apply, {M, F, A}}` 元组中调用 `application:start/1`，而是调用 `application:start_boot/2`。这是一个未公开的函数，与 `application:start/2` 不同，它假定目标 `application` 已被加载并要求 `application controller`（应用程序控制器）启动它。
6. 在发出最终的 `started` 进度报告之前，我们调用 `c:erlangrc()`。该函数没有对应的文档，它会读取并执行 `home` 目录或 `Erlang` 根目录中的 `.erlang` 文件。在该文件中设置代码路径和执行其他函数很方便。

请留心目标环境中的代码搜索路径。我们的示例可以启动 `bsc application` 的唯一原因是在启动 `Erlang` 时，在命令行提示符下使用 `-pa` 提供了 `beam` 文件的路径。我们的基站脚

本原本期待它们会位于 `$ROOT/lib/bsc-1.0/ebin`。在为目标环境生成启动脚本时，所有 `application` 都会被假定位于根目录的 `$ROOT/lib/` 下的 `AppName-version` 目录中。当我们生成目标目录结构和文件时，这一点将更加明显。

make_script 的参数

来更详细地看看我们可以传递给 `make_script/2` 调用的所有选项。我们已经知道 `Name` 是 `rel` 文件的名称：

```
systools:make_script(Name, OptionsList).
```

`OptionsList` 对应的选项包括如下几项。

src_tests

默认情况下，`systools` 假设 `beam` 文件是最新的并对应最新版的源代码。该标志会导致它去确认 `beam` 文件是否比相应的源文件更新，并且没有任何源文件找不到，否则会发出警告。

{path,DirList}

用于将 `DirList` 中列出的路径添加到代码搜索路径中。在启动 Erlang VM（用于执行 `systools` 函数）时，可以使用此选项，并且也可以传递参数 `-pa` 和 `-pz`。另外你可以在路径中包含通配符，因此 `"lib/*/ebin"` 将展开针对 `lib` 目录的每一个子目录，如果其包含名为 `ebin` 的子目录，则将被列入。

local

在启动脚本中放置本地路径而不是绝对路径。该标志非常适合需要在本地计算机上基于代码和 Erlang 运行时系统测试引导脚本的场景。

{variables,[{Prefix, Var}]}

用于将路径中的前缀替换为对应的变量。这使得你可以为一部分或所有 `application` 指定另一个目标路径。例如定义了前缀 `{"$BSC", "/usr/basestation/"}` 后，如果 `application` 和 `beam` 文件是在 `/usr/basestation/lib/bsc/ebin` 找到的，则生成的路径将会是 `$BSC/lib/bsc-1.0/ebin`。同样，如果本地路径为 `/usr/basestation/ernie/lib/bsc/ebin`，则生成的路径将会是 `$BSC/ernie/lib/bsc-1.0/ebin`。

{outdir, Dir}

将引导文件和脚本文件输出到 `Dir` 中。

exref 和 {exref, AppList}

使用 `Xref` 交叉引用工具测试发行包，该工具能够找出对未定义和已弃用函数的调用。

返回格式为 {ok, ReleaseScript, Module, Warnings} 或 {error, Module, Error} 的元组，而不是将结果打印到 I/O。当你从脚本调用 `systools` 函数或在自动构建过程中集成此调用并需要处理错误时使用此选项。

`no_dot_erlang`

删除用于加载和执行 `.erlang` 文件中表达式的指令。

`no_warn_sasl`

如果你不打算把 `sasl` 作为你的默认 application 之一，并且对生成的警告不感兴趣，则可以使用它。

`warnings_as_errors`

将警告视为错误，并在发生警告时拒绝生成脚本文件和引导文件。

备选引导文件

如果你查看当前正在运行的标准 Erlang/OTP 分发中的 `releases` 目录，会找到 4 个引导文件和 3 个 `rel` 文件。它们启动和加载的 application 是不同的。

`start_clean.boot`

按照 `start_clean.rel` 文件中的定义启动 `kernel` 和 `stdlib` application。

`start_sasl.boot`

按照 `start_sasl.rel` 文件中的定义启动 `kernel`、`stdlib` 和 `sasl` application。

`no_dot_erlang.boot`

启动 `kernel` 和 `stdlib` application，但不执行 `.erlang` 文件中的命令。当确定性很重要时，这非常有用，因为它不允许修改代码搜索路径以及修改其他用户首选项。

第 4 个文件 `start.boot` 是在安装 Erlang 时选择上述文件中的某一个而得到的副本，其作为默认文件。如果你想试用，可以在 `releases` 目录中将上述三个文件中的任何一个重命名为 `start.boot`。

你可以编写自己的脚本文件，或使用 `systools:make_script/2` 生成，或更改现有的脚本文件。如果你需要从脚本文件中生成发行包引导文件，请使用 `systools:script2boot(File)` 函数。

在过去，当需要调试启动问题时，就得更更改脚本文件。为了准确找出问题发生的位置，我们必须在每项操作后添加进度报告。当面对包含数千个模块的项目时，如果目标计算

机上安装的某个 beam 文件在构建或传输过程中遭到破坏，找到它的唯一方法就是在启动文件中的每个 `primLoad` 命令之后添加进度报告。它告诉我们在哪个 `application` 目录中存在问题，之后我们逐个加载所有模块，即可找到罪魁祸首。

如今，你可以通过将 `-init_debug` 标志传递给 `erl` 命令来启动跟踪功能。这将使得启动阶段更加可见。当用户不知道这个选项时，调试启动错误最终可能比在大海中捞针更难。虽然存在风险，但修改和编写自己的发行包文件仍然是有理由的：为了缩短启动时间而只加载特定模块和只启动特定 `application` 或更改其启动顺序等。

打包发行包

我们已经了解了创建与启动简单目标系统时的部分细节，并制作好了一个启动文件，到了该学习专家们是如何打包、部署、启动发行包的时候了。部署 Erlang 节点的最可靠和灵活的方式是采用嵌入式目标系统（`embedded target system`）。关于“嵌入式”的含义我们会在本章中进行解释，然而不幸的是，Erlang/OTP 中有好几种不同的上下文中都会使用这一术语，如果不留心区别将会搞混，所以请不要以为每次我们使用它时都意味着同一件事。此处，我们使用“嵌入式”一词，指的是目标系统将会成为在底层操作系统和硬件上运行的更大程序包的一部分。它能够在后台作为守护进程执行，无须启动交互式 `shell` 或始终保持开启状态，并且通常在操作系统启动时启动。为了在没有 `shell` 的情况下与其通信，嵌入式目标系统会通过管道传输所有 I/O。

由于设计和运营需要的不同，目标环境也会随之变化，因此无法提供万用的解决方案。下面描述了打包发行包的基本步骤，但在实践中，你可能需要根据实际情况进行一些调整：

1. 创建目标目录及 `rel` 文件。
2. 创建 `lib` 目录，并在其中放置与 `rel` 文件中所指定的版本匹配的 `application`。
3. 创建 `release` 目录，并在其中准备好启动脚本和 `application` 配置文件。
4. 将 `erts` 可执行文件和相关二进制文件复制到目标目录。
5. 创建 `bin` 目录并将配置文件和启动脚本复制到其中。

上述步骤（的全部或至少一部分）可以整合到自动构建系统中，生成安装脚本放在目标系统上执行，或是通过其他工具完成。哪些操作放在构建环境（`build environment`）中完成，哪些操作又放在目标环境（`target environment`）下借助安装脚本完成，以及哪些操作通过手工完成，哪些通过自动化完成等问题，不同的用户有不同的选择，原因是起初 OTP 未附带用于创建目标发行包（`target release`）的工具，但后来又包含了一个专注于批处理（`batch handling`）的复杂的工具。如果你构建时使用的是与目标环境相同的硬件和操

作系统，那么最好将操作统一放在一个地方完成。否则，如果你不确定最终的部署目标环境，或者需要针对不同目标环境动态创建有针对性的配置文件，那你恐怕必须得把某些步骤留到在目标环境上安装时执行。

我们现在会采用手工方式一步步创建出目标系统，并假设我们的开发环境和目标环境一致。根据你过去的构建、部署和配置经验，应当能够轻松划分出哪些操作应放在构建过程中，哪些应放在目标环境中。另外，我们还将介绍一些可用于自动执行此过程的工具。

我们首先创建目标目录 *ernie*^{注1}，然后将 *releases* 和 *lib* 目录添加到其中。不但需要使用某些标准的 UNIX 命令，还会用到 `systools:make_tar/2` 库函数。我们会从 *bsc application* 所在的目录下启动。注意，`make_tar` 调用期望 *rel* 文件、引导文件以及配置文件也位于此目录下。

在这个阶段，配置文件是可选的。你可能希望在安装时针对不同目标环境设定不同的值，以覆盖 *app* 文件中所指定的值。如果你在此阶段略去了配置文件，那么在安装系统时可不要忘了添加，否则系统将无法启动。配置文件必须命名为 *sys.config*，但你可以通过更改启动时传递给虚拟机的参数采用其他名字。

创建我们的 *ernie* 目标目录，重命名配置文件 *sys.config*，并将其放置在与 *bsc application* 以及 *rel* 和引导文件相同的目录中。完成这些准备工作后，就可以创建我们的 *tar* 文件了：

```
$ mkdir ernie
$ cp bsc.config sys.config
$ erl
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
1> systools:make_tar("basestation",
                    [{erts, "/usr/local/lib/erlang/"},
                     {path, ["bsc/ebin"]}, {outdir, "ernie"}]).
```

285

```
ok
2> q().
ok
3>
$ cd ernie
$ ls
basestation.tar.gz
$ tar xf basestation.tar.gz
$ ls
```

注1 *ernie* 是在 AXD301 ATM 交换机上运行 Erlang 节点所采用的用户名——它唤起了早期以各种形式对 Erlang 做出过贡献的人的美好回忆，包括本书的许多审校者。


```

basestation.tar.gz      lib
erts-7.2                releases
$ ls lib/
bsc-1.0                 kernel-4.1.1    sasl-2.6.1      stdlib-2.7
$ ls releases/
1.0                     basestation.rel
$ ls releases/1.0/
basestation.rel start.boot      sys.config
$ ls erts-7.2/bin/
beam          dialyzer    erl.src      heart        start.src
beam.smp      dyn_erl    erlc         inet_gethost start_erl.src
child_setup   epmd      erlexec      run_erl      to_erl
ct_run        erl       escript      start        typer
$ rm basestation.tar.gz

```

我们调用 `systools:make_tar(Name, OptionsList)` 生成了 *basestation.tar.gz* 包。Name 是发行包的名称，OptionsList 包含所有 `make_script` 接受的参数，并包括 {erts, Dir} 指令。当我们希望包含运行时系统二进制文件时，可以给出这个指令，以便生成 *erts-7.2* 目录。但我们想要的未必总是这样，比如运行时系统二进制文件可能已经安装在目标机器上，或者希望多个节点共用同一个版本的 Erlang 来运行等。此外还请注意，*sys.config* 文件包含在 *releases/1.0* 目录中。如果它的位置与 *rel* 文件所处的目录不同，则必须在安装的稍后阶段复制它。

你既可以将 *basestation.tar.gz* 部署到目标机器上，并在安装节点时运行本地配置脚本，也可以在构建环境中执行此操作，然后创建单个 tar 文件，用于部署此种节点。请记住，你的节点可能运行数万个独立安装——你们公司销售出的每一份基站控制器对应一个——或者，也可能在单台主机上，基于同一份安装，启动多个节点。配置参数将取决于你的需求以及安装的类型；既有可能这数以万计的部署都使用相同的配置参数，也有可能针对每种目标环境有不同的个性化定制。一般而言，往往是这两种策略的组合。另外，配置脚本既可以专用于你的系统，并且包含在 tar 文件中；还可以借由第三方部署和配置工具（如 Chef、Puppet 或 Capistrano）进行管理。

在本例中，我们手动解压了 *basestation.tar.gz* 文件。至此，之后剩下的步骤你可以选择在目标环境中运行，也可以选择在构建环境中运行。解压文件后，我们找到三个新目录：*lib* 目录（包含了所有 application 目录，并各自带有版本号）、*releases* 目录和 *erts* 目录。*erts* 目录之所以会出现，是因为我们在调用 `sys_tools:make_tar/2` 时包含了 {erts, Dir} 指令。

我们已经知道 Name 是 rel 文件的名称：

```
systools:make_tar(Name, OptionsList).
```

◀ 286

OptionsList 是可以为空列表或包含以下元素的某种组合：

`{dirs, IncDirList}`

复制时也包含 application 下指定的子目录（及默认的 *priv* 和 *ebin* 目录）。因此，若想将 *tests*、*src* 和 *examples* 也添加到发行包中，请将 *IncDirList* 设置为 [*tests*, *src*, *examples*]。

`{path, DirList}`

将路径添加到代码搜索路径中。此选项可以与启动运行系统的 Erlang VM 时传递的参数 *-pa* 和 *-pz* 一起使用。还可以在路径中包含通配符，例如，["*lib/*/ebin*"] 将展开 *lib* 中包含 *ebin* 目录的所有子目录。

`{erts, Dir}`

将目录 *Dir* 中找到的 Erlang 运行时系统的二进制文件包含到 tar 文件中。运行时系统的版本号是从 *rel* 文件中提取的。请确保该二进制文件已在目标操作系统和硬件平台上编译和测试过。

`{outdir, Dir}`

将 tar 文件输出到目录 *Dir* 中。如果省略此选项，默认会输出至 *rel* 文件所在的目录。

exref 和 `{exref, AppList}`

使用 *Xref* 交叉引用工具测试发行包，该工具能查找出对未定义函数或弃用函数的调用。这与传递相同选项给 *systools:make_script/2* 调用时所执行的测试相同。

src_tests

如果源代码和 *beam* 文件之间存在差异，则发出警告。这与传递相同选项给 *systools:make_script/2* 调用时所执行的测试相同。

287 **silent**

返回格式为 `{ok, ReleaseScript, Module, Warnings}` 或 `{error, Module, Error}` 的元组，而不是将结果打印到 I/O。你可以调用 `Module:format_error(Error)` 和 `Module:format(Warning)` 以分别获取格式化过的错误和警告信息。如果你在构建进程中集成了 *systools*，则请使用此选项；它的工作方式与传递相同选项给 *systools:make_script/2* 时相同。

另外两个选项 `{variables, [{Prefix, Var}]}` 和 `{var_tar, VarTar}` 允许你更改和操控创建目标库和包的方式。当需要做一些不同于标准 Erlang 的事时可使用它们，例如，如果你希望将发行包部署为 *deb*、*pkg*、*rpm* 或其他包或容器，它们允许你改变 application 的安装位置（默认为 *lib* 目录），并影响包的存储位置和存储方式。本章不使用这些选项，

更多信息和一些示例，请阅读 `systools` 参考手册页。

启动脚本以及目标上的配置

现在我们准备好了目标文件，接下来需要配置启动脚本。此处我们通过手动完成这些步骤，稍后会介绍可自动执行此过程的工具：

1. 在目标目录（在我们的例子中为 *ernie*）中，创建一个 *bin* 目录，在该目录中创建启动脚本并编辑好，它们将用于引导我们的系统。
2. 创建 *log* 目录，将启动脚本的所有调试输出存储到该目录。当系统无法启动时此处将是首先要查看的地方。
3. 在 *releases* 目录中创建一个名为 *start_ertl.data* 的文件，其中包含 Erlang 运行时系统及其发行包的版本号。
4. 如果原始 tar 文件中不包含 *sys.config* 文件，请创建一个（可能为空）并将其放入发行版本目录中。

此刻，十指交叉，一切即将启动。让我们更细致地过一遍这些步骤，重点是我们添加和编辑的那些文件。它们都在 *ernie* 目录中：

```
$ mkdir bin
$ cp erts-7.2/bin/start.src bin/start
$ cp erts-7.2/bin/start_ertl.src bin/start_ertl
$ cp erts-7.2/bin/run_ertl bin
$ cp erts-7.2/bin/to_ertl bin
$ mkdir log
```

在我们的例子中，我们创建 *bin* 目录并复制 *start.src* 和 *start_ertl.src* 到其中，分别将它们重命名为 *start* 和 *start_ertl*。我们还复制了 *run_ertl*，因为 *start* 脚本期待与其存在于同一目录，而复制 *to_ertl* 则是因为我们需要使用它来连接到嵌入式 Erlang shell。*start* 脚本负责初始化嵌入式系统的环境，然后调用 *start_ertl*，而 *start_ertl* 又会使用 *run_ertl* 脚本启动 Erlang。

288

将 *start_ertl* 视为嵌入式版本的 *ertl*，而 *start* 则是供你随心自定义的脚本。根据你的需求和要求，你可能还需要属于你的 *ertl* 和 *heart* 脚本，并且如果打算运行分布式 Erlang，则还需要 *epmd* 二进制文件。所有这些都可以从运行时系统的 *bin* 目录复制而来。

文件和二进制文件都准备好后，我们需要修改它们。需要修改 *start* 文件，用新的 Erlang 根目录的绝对路径替换 `%FINAL_ROOTDIR%`。在我们的例子中，这个目录是 *ernie*，使用 `perl` 及其 `-i` 就地修改选项来更改文件，使用 `shell` 的 `PWD` 变量的值替换文本。然后，我

们使用 `diff` 命令展示了修改前后的差别：

```
$ pwd
/Users/francescoc/ernie
$ perl -i -pe "s#%FINAL_ROOTDIR%#%PWD#" bin/start
$ diff erts-7.2/bin/start.src bin/start
27c27
< ROOTDIR=%FINAL_ROOTDIR%
---
> ROOTDIR=/Users/francescoc/ernie
$ echo '7.2 1.0' > releases/start_erl.data
$ bin/start
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.1 (^D to exit)
```

```
1> application:which_applications().
[{bsc,"Base Station Controller","1.0"},
 {sasl,"SASL CXC 138 11","2.6.1"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]
2> [Quit]
$ ls /tmp/erlang.*
/tmp/erlang.pipe.1.r /tmp/erlang.pipe.1.w
```

修改了 `start` 文件后，我们在 `releases` 目录中创建了 `start_erl.data` 文件。它包含 Erlang 运行时系统与发行包对应的版本（这两项在我们的示例中都是用空格分隔的），并且 `release` 目录下包含了所有的引导脚本和配置文件。

我们现在可以使用 `start` 命令启动系统了。请注意，与之前使用 `erl` 命令不同，此发行包是以后台作业方式启动的。要连接到它的 Erlang shell，我们使用 `to_erl` 命令，并将 `/tmp` 目录传递给它作为管道读写的场所。



在运行嵌入式 Erlang 系统中，你可能会习惯地使用 `Ctrl+C` 命令退出 shell。`Ctrl+C` 调用虚拟机中断处理程序，之后可以执行以下命令之一：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
        (v)ersion (k)ill (D)b-tables (d)istribution
```

如上所示，`a` 终止 Erlang 节点。

为避免终止节点，可使用 `Ctrl+D` 退出 shell。如果你出于习惯输入 `q()`、`halt()` 或 `Ctrl+C`，将终止整个后台作业。通过使用 `Ctrl+D`，你可以退出 `to_erl` shell 的同时保持 Erlang VM 在后台继续运行。

如果你尝试连接到计算机上的管道时碰到错误提示 *No running Erlang on pipe /tmp/erlang.pipe: No such file or directory*, 请查看 *log* 目录以了解 Erlang 节点无法启动的原因。脚本中的所有启动错误都将记录在那里。问题可能包括路径错误、缺少 *sys.config* 文件、启动文件损坏或二进制文件命名错误等。

在目标系统的 *bin* 目录中包含 *erl* 命令是一种好习惯。当遇到出现某些错误无法重新启动节点的状况时, 这将为你带来希望。在出现错误的场景下, 第一件要做的事就是查看 SASL 报告日志, 崩溃和错误报告在大多数情况下会告诉你引发节点故障的错误链。你最不愿看到的事情是每次你想查看 SASL 日志, 都得反复将 SASL 日志移动到其他远程计算机上查看, 原因是本机中的 Erlang 无法启动。请务必防患于未然, 始终生成类似于 *start_sasl.boot* 用途的第二个引导文件, 其中包含与你的系统版本相同的 *kernel*、*stdlib* 和 *sasl application*。

在我们的例子中, 我们使用 */tmp* 目录作为读写管道, 因为它是脚本文件使用的默认目录。但是, 如果你计划在同一台机器上运行多个嵌入式节点, 则会出现问题。一个好的做法是将管道重定向到目标结构目录中 Erlang 根目录的某个子目录。这允许多个节点实例在同一台计算机上运行, 这在许多系统中都是常见的做法。如果你查看启动脚本的最后一行, 会看到将 */tmp/* 替换为了根目录中新管道的绝对路径的位置。你还可以将所有日志重定向到其他地方:

```
$ROOTDIR/bin/run_erl -daemon /tmp/ $ROOTDIR/log "exec ..."
```

参数和标志

290

到目前为止还不错。如果我们想要启动一个分布式的 Erlang 节点或将一个 *patches* 目录添加到代码搜索路径该怎么办呢? 或者当我们已经写好了一个配置文件, 但不打算使用 *sys.config* 这一文件名, 而是希望沿用原来的 *bsc.config* 文件名该怎么办? 又或者, 更重要的一点, 是否有可以传递给模拟器的标志使得我们可以禁用通过 *Ctrl+C* 杀死节点的能力?

启动 Erlang 时, 我们可以将三种不同类型的参数传递给运行时系统。它们分别是模拟器标志 (emulator flag)、一般标志 (flag) 和素参数 (plain argument)。你可以通过其起始的 + 字符识别模拟器标志。它们控制虚拟机的行为, 允许配置系统限制、内存管理选项、调度程序选项以及针对模拟器的其他各个方面。

以 - 开头的标志会被传递给运行时系统中的 Erlang 部分。此类标志包括代码搜索路径、配置文件、环境变量、分布式 Erlang 设置等。

素参数是用户自己定义的, 与运行时系统无关。第一次见到它们是在第 9 章的“环境变量”

一节，当时用它们来覆盖 app 文件和配置文件中的 application 环境变量。此外还可以在 application 业务逻辑中使用素参数。

下面示例中的命令使用了若干个参数：

```
erl -pa patches -boot basestation -config bsc -init_debug +Bc
```

这会启动 Erlang 并将 *patches* 目录添加到代码搜索路径的开头。它还使用 *basestation.boot* 和 *bsc.config* 文件启动系统，并设置 *init_debug* 标志，从而在启动时增加调试消息的数量。*+Bc* 虚拟机标志会禁用 shell 中断处理程序，因此，当顺序按下 Ctrl+C A 时，不是终止虚拟机，而是终止 shell 进程并重新启动它。

让我们更详细地看看其中的一些虚拟机标志。我们选择了不涉及内存管理、多核架构、端口和套接字、低级跟踪或其他内部优化的高级标志和系统限制标志。内部优化不在本书的讨论范围之内，只有在清楚自己在做什么的情况下才可应谨慎使用。你可以在 *erl* 的手册页面阅读更多我们已经介绍过的（以及更多未介绍过的）内容。我们下面列出的这些，并不是针对系统优化的，而是日常常用的（在你还没想着优化之前就已经会用到的）。

+Bc

对于在线系统，开启中断处理程序（break handler）是危险的，因为你的手指往往先于你的意识（特别是如果你半夜被叫起来做维护，你的脑袋还处于昏昏欲睡状态时）。如果你曾经以此方式终止过 shell，同样也会不小心在生产系统上这么做。使用 *+Bc* 标志可以使你用 Ctrl+C A 时终止当前 shell 并启动一个新的 shell，不会影响到系统。你的所有线上系统都应当开启此选项。

+Bd

此选项使你可以使用 Ctrl+C 直接终止 Erlang 节点，完全绕过中断处理程序。

+Bi

此选项使模拟器完全忽略 Ctrl+C，在这种情况下，终止 Erlang 虚拟机的唯一方法是使用 shell 命令 *q()* 或 *halt()* BIF。这个选项很危险，因为如果交互式调用无法返回，你就无法恢复了，结果导致 shell 被挂起。

+e Num

设置 ETS 表的数量上限，默认是 2053。如果你使用的是 Erlang/OTP R16B03 或更新的版本，可以在运行过程中通过调用 *erlang:system_info(ets_limit)* 来获得系统 ETS 表数量的上限值。

+P Num

更改系统中允许同时存在的进程数上限。默认限制为 262 144，但你可以设置的范

围是 1024 到 134 217 727。

+Q Num

更改系统允许的端口数上限，默认为 65 536。可设置的范围是 1024 到 134 217 727。

+t Num

更改系统允许的原子数量上限，默认为 1 048 576。这些限制是针对 Erlang 17 及以上版本的，并且是基于 UNIX 的操作系统。其他操作系统上的默认值可能不同。

+R Rel

使得你的 Erlang 节点能够以分布式 Erlang 方式连接到其他运行着（可能非后向兼容的）老版本的分布式协议的节点。

普通的标志是在启动时定义的，并在运行过程中由系统中的 Erlang 部分获取，并由标准的及用户定义的 OTP application 使用。请记住，大部分 Erlang 核心及运行时系统都是用 Erlang 编写的，因此在 application 中定义和检索这些标志的方法与 Erlang 自身在运行系统内所采用的定义和检索它们的方式是相同的。以下列出了主要的一些标志。

-Application Key Value

将 *Application* 的名为 *Key* 的环境变量设置为 *Value*。我们曾在第 9 章的“环境变量”一节中介绍过该选项。

-args_file FileName

允许你把所有的普通标志、模拟器标志、素参数都放到名为 *FileName* 的单独的配置文件中，模拟器启动时将会读取该文件。该文件还可以包含以 # 字符开头的单行注释。使用一个单独的文件来放参数是比较好的，可以避免与启动脚本中需要设置或更改的参数搞混。这种方法还可以让你将参数文件纳入版本控制，与代码的其余部分保持一致。

292

-async_shell_start

允许 shell 与系统的其他部分并行启动，而不是默认在系统完全启动之前不处理在 shell 中键入的任何内容。当你尝试调试启动问题或找出超时发生的位置时，这非常有用。

-boot filename

指定引导文件的名称为 *filename.boot*。如果提供的不是绝对路径，则模拟器会假定引导文件位于 *\$ROOT/bin* 目录中。

-config filename

指定配置文件的位置和名称为 *filename.config*。

-connect_all false

阻止 global 子系统维护一个全连接的分布式 Erlang 节点网络，这样做实际上禁用了该子系统。

-detached

以与系统控制台分离的方式启动 Erlang 运行时系统。当你想将其作为守护程序和后台进程运行时會用到此选项。-detached 选项意味着 -noinput，即启动 Erlang 节点，但不是运行能够解释执行你输入的所有命令的读取 - 求值循环的 shell 进程。使用 -noinput 的同时也意味着使用了 -noshell 命令，该命令启动不带 shell 的 Erlang 运行时系统，这样做的一种潜在的场景是作为 UNIX 管道序列中的一个环节执行。

-emu_args

在启动时打印传递给模拟器的所有参数。在你的生产系统中应总是使用这一选项，因为你永远不知道何时会需要查阅这些信息。

293

-init_debug

在启动时为你提供详细的调试信息，展示引导脚本中执行的每个步骤。使用 -init_debug 和 -emu_args 的开销可以忽略不计，但它们提供的信息在进行故障排除时可以说是无价之宝。

-env Variable Value

另一种方便的用于设置操作系统环境变量的方式。它主要用于测试，但在针对某些 Erlang 值做处理时也很有用。

-eval

在节点初始化过程中，解析并执行指定的 Erlang 表达式。如果解析或执行出错，则导致该节点关闭。

-hidden

当使用分布式 Erlang 时，将当前 Erlang 运行时系统作为隐藏节点启动，既不告知外部自身的存在，也不告知外部自身所连接的其他节点的存在。

!-heart

启动针对 Erlang 运行时系统的一个外部监视器。如果发现被监视的虚拟机终止了，则将调用某个指定的脚本进行重启。我们在本章后面的“heart”一节中详细介绍了这一机制。

-mode Mode

规定了系统加载代码的方式。如果 *Mode* 是 *interactive*，则调用一个尚未加载的模块，会自动根据代码搜索路径查找。你的目标系统应该以 *embedded* 模式运行，这种模式下所有模块都会在引导文件启动时就加载，并且调用任何未加载的模块都会导致崩溃。不过，在 *embedded* 模式下你仍然可以在 *shell* 中使用 `l(Module)` 或 `code:load_file(Module)` 调用来加载模块。

建议在所有生产系统中都采用 *embedded* 模式。它能够保证当你的某个进程发起关键调用时，不会因为需要查找未加载的模块而遍历代码搜索路径导致进程暂停。

-nostick

禁用一种特殊的机制，该机制用于避免加载和覆盖不可动 (*sticky*) 目录中的模块。默认情况下，*kernel*、*compiler* 和 *stdlib* 这三个 application 的 *ebin* 目录都被认为是不可动的 (*sticky*)，这是一种手段，旨在防止系统的关键元素被意外损坏。

-pa 和 -pz

在这两个命令中，前者用于将包含 *beam* 文件的目录添加到代码搜索路径的开头；后者则是添加到结尾。一种常见的用法是添加 `-pa patches` 以指向针对用于为发行包提供临时修补程序的目录。

-remsh node

使用分布式 Erlang 方式启动一个 *shell* 连接到远程节点 *node*。当你是以无 *shell* 方式运行节点时，或是需要远程连接到节点时非常有用。

-shutdown_time Time

指定允许系统花费多少时间用于关闭监督树，单位是毫秒。默认值为无穷大。不过，请谨慎使用此选项，因为它会覆盖 *behavior* 子进程规范中指定的关闭值。

-name name 和 -sname name

在使用分布式 Erlang 时，这两个命令的作用分别是启动具有长名和短名的分布式节点。如果节点间要相互通信，它们必须共享同一个 *cookie*，其值可以使用 `-setcookie` 指令设置，并且所有 *cookie* 都有长名或短名。短名节点和长名节点间不能相互通信。

-s module、-s module function、-s module function args

这些格式中的第一种是在启动时执行 `module:start()`。第二种则是执行 `module:function()`。第三种就像第二种，但包含了函数的参数列表。所有的 *args* 都会被作为原子进行传递。`-run` 选项的工作原理与此类似，不同之处在于，如果定义了参数，它们将作为字符串列表传递给 `module:function/1`。由 `-run` 和 `-s` 执行

的函数必须返回，否则将导致整个启动过程卡住。而如果它们异常终止，也会导致节点终止，并中断启动过程。

当需要排除系统故障时，可以使用分布式 Erlang 连接到远程节点。例如，假设你想连接到节点 `foo@ramone`，该节点的 cookie 为 `abc123`。你可以通过启动一个带有 `-remsh` 标志的 Erlang 虚拟机来实现：

```
$ erl -sname bar -remsh foo@ramone -setcookie abc123
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
(foo@ramone)1> node().
```

```
foo@ramone
```

```
(foo@ramone)2> nodes().
```

```
[bar@ramone]
```

```
(foo@ramone)3>
```

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
        (v)ersion (k)ill (D)b-tables (d)istribution
```

```
a
```

```
$
```

所有命令将在 `foo` 中远程执行，结果则显示在本地。当你要退出本地 shell 时千万要小心。使用 `halt()` 和 `q()` 将终止远程节点。请始终使用 `Ctrl+C`。

现在让我们尝试在 shell 中使用 `-s`、`-eval` 和 `-run`，以了解它们的工作方式：

```
$ erl -s observer
```

```
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> q().
```

```
ok
```

```
$ erl -noshell \
```

```
-eval 'Average = (1+2+3)/3, io:format("~p~n",[Average]), erlang:halt()'
```

```
2.0
```

```
$ erl -run io format 1 2 3
```

```
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
123Eshell V7.2 (abort with ^G)
```

```
1> q().
```

```
ok
```

```
$ erl -s io format 1 2 3
```

```
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```



```

{"init terminating in do_boot",
 {badarg,[{io,format,[<0.24.0>,['1','2','3'],[],[]],{init,start_it,1,[]},{init,start_em,1,[]]}}}

```

```

Crash dump is being written to: erl_crash.dump...done
init terminating in do_boot ()

```

我们成功地使用 `erl -s observer` 调用了 `observer:start()`。你在这里显示的 shell 输出中没有看到它，但它确实打开了一个观察者（observer）`wxWidgets` 窗口。这是一种很有用的技巧，可以在启动虚拟机时一并启动调试工具。然后我们使用 `-eval` 标志计算了三个整数的平均值，打印结果并停止模拟器，所有这些都不需要启动 Erlang shell。在第三个和第四个例子中，我们使用 `-run io format 1 2 3` 调用 `io:format(["1","2","3"])`，使用 `-s io format 1 2 3` 调用 `io:format(['1','2','3'])`。后者崩溃是因为 `io:format/1` 期望的是字符串而传入的是原子列表。

使用 `-run` 和 `-s` 标志时，如果调用的是 `spawn_link` 和 `start_link` 要小心，这些函数会将自身链接到初始化进程，然而该进程是用于初始化系统的而不适合作为父进程。尽管当前进程在执行初始化调用后仍继续运行，但不应该依赖于这一行为，因为它没有文档化，并可能在将来的发行包中有所更改。

在 application 中可以使用 `init:get_arguments()` 和 `init:get_argument(Flag)` 函数来获取标志。Flag 可以是预定义的标志，如 `root`、`programe` 和 `home` 等，也可以是任何用户自定义的命令行标志。

296

在虚拟机标志和常规标志之前，在 `-extra` 标志之后，以及在 `--` 指令和下一个标志之间设定的所有参数都被认为是素参数。我们可以使用 `init:get_plain_arguments/0` 调用来获取这些素参数：

```

$ erl one -two three -pa bin/bsc -- four five -extra 6 7 eight
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

```

```

Eshell V7.2 (abort with ^G)
1> init:get_plain_arguments().
["one","four","five","6","7","eight"]
2> init:get_argument(two).
{ok,["three"]}
3> init:get_argument(pa).
{ok,["bin/bsc"]}
4> init:get_argument(programe).
{ok,["erl"]}
5> init:get_argument(root).

```

```
{ok, ["/usr/local/lib/erlang"]]}
6> init:get_argument(home).
{ok, ["/Users/francescoc"]}}
```

heart

通常会将嵌入式 Erlang 系统作为守护进程作业运行，并且会将其配置为随计算机重新启动而自动启动。这意味着如果发生停电或其他故障，或执行需要重启的维护操作，系统将自动启动。但是如果只有 Erlang 节点本身崩溃或停止响应会发生什么？造成这一问题的原因有很多，例如意想不到的内存高峰、顶级监督者终止、糟糕的 NIF 导致了虚拟机中出现分段错误等，甚至是某种虚拟机内部的罕见的错误导致其挂起。这就是为什么你需要启用 *heart*。你可以将 *heart* 视为节点本身的监督者。

heart 是一个外部程序，用于监视虚拟机，它负责接收 Erlang 进程通过端口定期发送的心跳信号。如果该外部程序在预定义的时间间隔内未能收到心跳信号，则它将尝试终止虚拟机并调用用户预先定义好的命令来重新启动运行时系统。

让我们来写一个简单的脚本 *bsc_heart*，它的作用很简单，仅仅是调用 *bin/start* 命令而已。我们确实可以直接将 *start* 设置为 *heart* 命令，但是仅盲目地重启往往无法应对实际中的复杂情况，因此通常会使用重启脚本。这样一来，我们有机会做到，在尝试重启失败后，做出推测：可能已陷入循环重启，而我们无法恢复这种故障，因此选择停止尝试重启节点。或者，在一定次数的重新启动尝试后，允许在一个递增变化的时间间隔内，重新启动操作系统。或者我们可以触发其他自动诊断脚本来对周围环境做健康测试。可用的方案有很多，具体如何选择取决于你的部署环境和监视 / 警报设施，因此重新启动脚本可以像你希望的那样很简单或很复杂。在此处，我们使用以下 *bsc_heart* 脚本，并将其放在目标安装目录的 *bin* 目录中：

```
#!/bin/sh
#Basic Heart Script for the Base Station Controller

ROOTDIR=/Users/francescoc/ernie

$ROOTDIR/bin/start
```

然后，设置 *HEART_COMMAND* 环境变量来调用此脚本，编辑 *start_erl* 脚本，使其包含 *-heart*，然后启动基站控制器。接下来我们试着以各种方式杀死它。系统虽然被我们杀死，但每次连接到 I/O 管道时，它都已再次启动并运行着了：

```
$ $ cp bsc_heart ernie/bin/.
$ export HEART_COMMAND=/Users/francescoc/ernie/bin/bsc_heart
$ vim bin/start_erl
```



```

$ diff erts-7.2/bin/start_erl.src bin/start_erl
47c47
< exec $BINDIR/erlexec ... -config $RELDIR/$VSN/sys ${1+"$@"}
---
> exec $BINDIR/erlexec ... -config $RELDIR/$VSN/sys ${1+"$@"} -heart
$ bin/start
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.5 (^D to exit)

1> halt().
heart: Sat Aug 23 12:49:47 2014: Erlang has closed.
[End]
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.5 (^D to exit)

1>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
[End]
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.5 (^D to exit)

1>

```

我们看到使用 `halt()` 或 `Ctrl+C` 会杀死节点，证据是每次连接时，命令提示符都是 1。`heart` 系统会立即重新启动该进程。

下述操作系统环境变量（全部为可选项）既可以在启动脚本中使用 `-env` 标志在引导系统阶段进行设置，也可以选择其他位置或时机进行设置：

298

HEART_COMMAND

发生超时时要执行的脚本名称。如果此变量未设置，则发生超时将仅触发警告，声称系统将被重新启动，然而系统并不会重新启动。

HEART_BEAT_TIMEOUT

在终止虚拟机并调用 `heart` 命令之前，等待心跳的秒数。在 Erlang 17 或更新版本中，它可以是大于 10 且小于或等于 65 535 的值。省略此设置会将超时默认视为 60 秒。

ERL_CRASH_DUMP_SECONDS

允许虚拟机在被杀死和重新启动前花费多少时间来写入崩溃转储文件。由于崩溃转储文件可能很大，虚拟机可能需要花一点时间才能将它们写入磁盘。使用 `heart` 而不设置此变量时的默认设置是 0，这意味着不会写入故障转储文件；该虚拟机立即死亡，并立即调用 `heart` 命令。将该值设置为 -1（或任何其他负数）允许虚拟机花

费任意长的时间直到完成故障转储文件写入。任何其他正整数均表示允许虚拟机在崩溃转储文件写入方面花费的时间，超出则被强制终止并重新启动。

在我们的例子中，我们决定直接采用在 UNIX shell 中设置环境变量的方式，也可以轻松地编辑 `start_erl` 文件或使用 `-env variable value` 参数将它们作为标志传递给 `erl`：

```
erl -heart -env HEART_BEAT_TIMEOUT 10
    -env HEART_COMMAND boot_bsc
```

299



使用 *heart* 时，要注意心跳信号机制以及重启二者之间存在竞态条件。如果你没有预料到并检查这些可能，当问题出现时将令你抓狂。曾经在一些案例中，Erlang 虚拟机由于承受了极端负载导致运行迟缓，同时即使发出了心跳信号，但由于操作系统的一些潜在问题，导致这些心跳信号一直无法抵达 *heart*，产生这一现象的原因，可能是由于 I/O 饥饿以及设置的 `HEART_BEAT_TIMEOUT` 值过低。由于心跳无法抵达，因而 *heart* 终止了 Erlang 虚拟机并重启它。但这一过程不会生成崩溃转储文件，因为 *heart* 终止虚拟机所采用的是 `SIGKILL` 方式，这种方式（至少在类 UNIX 系统上）无法被目标捕捉处理。杀死 Erlang 虚拟机进程（并重新启动操作系统，如果需要）是否能解决这个问题？也许能，但对想搞清楚状况的可怜的开发人员来说，当他们希望搞清楚到底发生了什么，却找不到 Erlang 虚拟机崩溃转储文件时，会感觉很无助。

heart 在大多数操作系统上都能正常运行。讨论它在 Windows 和其他非 UNIX 型操作系统上是如何执行的超出了本书的讨论范围，同样，探讨如何连接和配置以使得它能够使用 Solaris 的硬件看门狗定时器也超出了本书的讨论范围。有关这些方面的更多信息，请阅读标准 Erlang 分发附带的 *heart* 手册页。

Yaws 是如何使用 heart 的

作为 *heart* 用法的一个例子，让我们看一下 Yaws Web Server 是怎么使用它的，它最初是由 Claes “Klacke” Wikström 开发的一个项目，可以从 Yaws 官方站点 (<http://yaws.hyber.org/>) 以及 GitHub (<https://github.com/klacke/yaws>) 上获得。Yaws 以一种有趣的方式使用了 *heart*：为了避免 *heart* 的 stubborn 行为无休止地尝试重启目标，Yaws 的重启脚本会记录自身在一定时间内重启了多少次，这有点类似 OTP 中 supervisor 监视子进程重启次数那样。为了实现这一点，Yaws 设置了如下所示的 `HEART_COMMAND`：

```
HEART_COMMAND="$ENV_PGM \  
    HEART=true \  
    YAWS_HEART_RESTARTS=$restarts \  
    YAWS_HEART_START=$starttime \  
    $program"
```


如你所见，Yaws 的 `HEART_COMMAND` 值中包含了几个其他变量，这些变量将会在 `heart` 重启时由 shell 脚本处理：

HEART 环境变量

设置为 `true`，这样 Yaws 就知道 `heart` 正发挥着控制功能。

YAWS_HEART_RESTARTS 环境变量

追踪 Yaws 已经被重启了多少次。

YAWS_HEART_START 环境变量

追踪启动时间，基于 UNIX 纪元（1970 年 1 月 1 日以来的秒数）。

\$restarts 和 \$starttime shell 变量

根据上次重启时设置的 `YAWS_HEART_RESTARTS` 和 `YAWS_HEART_START` 值，Yaws 可以为 `HEART_COMMAND` 计算出新的设置。

当运行 Yaws 的时候，你可以通过命令行参数指定一个时间范围内允许重启的最大次数。如果 Yaws shell 脚本通过这些环境变量检测到已在设定的时间范围内重启了足够的次数，它将生成一条错误消息并且拒绝再重启。要想了解更多细节，可以看看 Yaws 启动脚本的源代码。

Erlang 模块加载器

有时你需要把发行包放在嵌入式设备上运行，这类环境往往只有很小的磁盘空间，甚至是无磁盘空间，因此你会希望改变运行时系统加载模块的机制。你不打算以文件的方式来读取模块，而是希望从数据库中加载，或者从网络中的另一个节点加载。使用 `-loader` 参数可以指定 `erl_prim_loader` 获取模块的方式。默认的加载器（loader）是 `efile`，它会从本地文件系统中加载模块。如果你想在另一台机器上使用引导服务器（boot server），必须指定使用 `inet` 加载器。当使用 `inet` 时，必须通过 `-id name` 参数指明引导服务器所处的远程节点的名称，此名称 `name` 对应的是你启动目标引导服务器时所传入的 `-name` 或 `-sname` 标志。你还必须把目标机器的 IP 地址也通过 `-hosts address` 标志传入，其中 `address` 是一个 IP 地址字符串，例如 4 个整数由点号分隔的形式。一个例子是 `-id foo -hosts "127.0.0.1"`，其指定了引导服务器是运行在本机上的名为 `foo` 的 Erlang 虚拟机中。

为了能更直观地了解加载过程，我们首先使用 `systools:make_script/2` 并传入 `local` 选项生成一个 `basestation.boot` 文件。`local` 选项很关键，因为它保障了 `bsc` 的 beam 文

件的本地副本能够被正确找到，而无须将它们安装到官方发行包的 *lib* 目录中。它的作用实际上是把 *bsc* application 的本地路径添加到了引导服务器的加载路径中，从而使得能够成功地生成 *basestation.boot* 文件：

```
$ erl -pa bsc/ebin
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V7.2 (abort with ^G)
1> systools:make_script("basestation", [local]).
ok
```

301 接下来，我们启动引导服务器：

```
$ erl -name foo@127.0.0.1 -setcookie cookie
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V7.2 (abort with ^G)
(foo@127.0.0.1)> erl_boot_server:start([{127,0,0,1}]).
{ok,<0.42.0>}
```

引导服务器启动并准备好处理请求后，我们可以启动 *bar* 节点：

```
$ erl -name bar@127.0.0.1 -id foo -hosts 127.0.0.1 \
    -loader inet -setcookie cookie \
    -init_debug -emu_args -boot basestation
Executing: /usr/local/lib/erlang/erts-7.2/bin/beam.smp
/usr/local/lib/erlang/erts-7.2/bin/beam.smp --
-root /usr/local/lib/erlang -progname erl --
-home /Users/francescoc --
-name bar@127.0.0.1 -id foo -hosts 127.0.0.1
-loader inet -setcookie cookie
-init_debug -boot basestation
{progress,preloaded}
{progress,kernel_load_completed}
{progress,modules_loaded}
{start,heart}
{start,error_logger}
{start,application_controller}
{progress,init_kernel_started}
...<snip>...

==PROGRESS REPORT==== 26-Dec-2015::12:59:05 ===
    supervisor: {local,bsc}
    started: [{pid,<0.50.0>},
```



```

        {id,simple_phone_sup},
        {mfargs,{simple_phone_sup,start_link,[]}},
        {restart_type,permanent},
        {shutdown,2000},
        {child_type,worker}]
{apply,{c,erlangrc,[]}}

```

=PROGRESS REPORT==== 26-Dec-2015::12:59:05 ===

```

    application: bsc
    started_at: 'bar@127.0.0.1'
{progress,started}
Eshell V7.2 (abort with ^G)
(bar@127.0.0.1)> application:which_applications().
[{bsc,"Base Station Controller","1.0"},
 {sas1,"SASL CXC 138 11","2.6.1"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]

```

正如输出所展示的那样，我们的节点能够借助远程的引导服务器而启动。尽管我们的例子中用的是本地主机（127.0.0.1），你可能会疑虑这样的机制是否同样适用于远程网络环境，不用担心，你可以在你的本地网络中使用两台不同的主机进行测试，亲眼看看相关文件是否真的能从远程引导服务器处加载得到。

◀ 302

init 模块

init 模块是预加载到 Erlang 运行时系统中的。它为你的发行包管理参数以及启动和关闭进程。在启动时，它会执行引导文件中的所有命令。我们感兴趣的地方在于，正是通过此模块，我们才获得了重新启动系统、干净地关闭所有 application、停止节点以及重新启动虚拟机的能力。以下是该模块的常用用途列表。

init:restart/0

在不重新启动模拟器的前提下重新启动 Erlang 节点。application 会被平滑地关闭、模块被卸载、端口被关闭，然后使用最初提供的相同引导参数再次执行引导文件。你可以使用 -shutdown_time 标志来限制停止 application 时允许花费的时间。

init:reboot/0

像 restart 一样，区别在于虚拟机也会被关闭并重新启动。heart（如果在使用的話）将尝试重新启动系统，这可能导致潜在的竞态条件，会在杀死虚拟机并重新启动时自行解决。使用 -shutdown_time 标志设置的超时值将被采纳。

`init:stop/0`

系统平滑关闭并停止虚拟机。如果 *heart* 正在运行，则在尝试重新启动节点之前，会先停止 *heart*。这是停止正在运行的节点的正确方法，因为它允许 application 自行终止并清理，并最终正确关闭。调用 `init:stop(Status)` 与调用 `halt(Status)` 具有相同的效果。使用 `-shutdown_time` 标志设置的超时值将被采纳。

`init:get_status()`

确定系统处于何种状态，正在启动、已经停止还是正在运行。它返回一个格式为 `{InternalStatus, ProvidedStatus}` 的元组，其中 `InternalStatus` 是 `starting`、`started` 或 `stopping` 之一。启动系统时，`ProvidedStatus` 表明当前正在运行启动脚本 `init` 中的哪个部分。它是从引导文件解释执行时所执行的最后一个 `{progress, Info}` 获得 `Info` 状态的。

303 我们已经在本章前面“参数和标志”一节中介绍了 `init` 模块中的其他有用功能，包括 `get_arguments/0`、`get_argument/1` 和 `get_plain_arguments/0`。

rebar3

本章介绍的许多手工操作都可以利用各种工具自动完成。当我们生成模板、构建发行包和生成目标结构时，都需要自动化操作。但因为目前还没有针对发行包制作制订标准——只有首选或推荐方案——因此现在随 Erlang / OTP 一起提供的工具并不能覆盖所有场景，需要社区开发的工具进行补充，这使得这些工具有时在功能上存在重叠。在本章的剩余部分，我们将介绍 *rebar3*，这是一个可以管理发行包和依赖关系的通用构建工具。

rebar3 工具是 *rebar* 工具的第二代，它是 Erlang 构建工具中使用最广泛的一种，是 Erlang 社区中最常用的工具之一。*rebar3* 是一个全面的工具，可以解决许多项目管理需求，包括依赖管理、编译和发行包生成等，并且你还可以通过插件增强或扩展其功能。

要获得 *rebar3*，可以从其网站下载预制好的版本：

```
$ curl -LO https://s3.amazonaws.com/rebar3/rebar3
```

或者克隆 *rebar3* 的 Git 仓库并从源代码构建它：

```
$ git clone https://github.com/erlang/rebar3.git
$ cd rebar3
$ ./bootstrap
====> Updating package registry...
...<snip>...
====> Compiling rebar
====> Building escript...
```


一些已经存在几年的 Erlang 项目仍然在其源代码库中包含了一份第一代的 *rebar* 可执行文件。这原本是为了让用户更容易构建项目而不强迫他们首先构建 *rebar*，但考虑到 *rebar* 已被普遍采用，继续遵循那样的传统，在你的项目中包含一份 *rebar3* 副本是过时而没有必要的。用户如今只需在 shell 路径的某处放置一个 *rebar3* 的副本，例如 */usr/local/bin*，然后从那里使用它即可。

不带参数运行 *rebar3* 将提供有关如何使用的信息。这是其输出的一部分：

```
$ rebar3
```

```
Rebar3 is a tool for working with Erlang projects.
```

```
Usage: rebar [-h] [-v] [<task>]
```

```
-h, --help      Print this help.
-v, --version    Show version information.
<task>          Task to run.
```

Several tasks are available:

```
...<snip>...
```

Run 'rebar3 help <TASK>' for details.

从这个输出中得到的是 *rebar3* 支持的任务列表。在此处完整展示该清单实在太长，但总体来看，*rebar3* 支持的任务分为以下几类。

构建命令

支持编译 Erlang 和非 Erlang 源代码并清理构建产出。

项目创建命令

基于模板生成骨架项目。

依赖管理命令

支持检索、构建、更新、清理和删除项目依赖。

发行包生成命令

支持创建发行包和升级包。

Test 命令

支持运行单元测试、*common_test* 套件和基于属性的测试。

rebar3 还提供了与其他各种项目活动相关的命令，例如文档生成、创建 *escript* 存档，以及启动一个 Erlang shell 并自动把所有项目文件和外部依赖文件都列入加载路径中等。

生成一个 *rebar3* 发行包项目

你可以使用 *rebar3* 并选择适当的项目模板来为我们的基站控制器示例系统生成项目骨架。虽然我们的示例只使用了一个用户定义的 *bsc* application，但我们采用的方案实际上同样能够适应多个 application 的情况，毕竟这是大多数项目的典型需求。

首先，我们创建一个新目录 *ernie2*，并在其中使用 *rebar3* 生成一个新的 *bsc* 发行包项目：

```
305 $ mkdir ernie2
$ cd ernie2
$ rebar3 new release bsc desc="Base Station Controller"
```

```
==> Writing bsc/apps/bsc/src/bsc_app.erl
==> Writing bsc/apps/bsc/src/bsc_sup.erl
==> Writing bsc/apps/bsc/src/bsc.app.src
==> Writing bsc/rebar.config
==> Writing bsc/config/sys.config
==> Writing bsc/config/vm.args
==> Writing bsc/.gitignore
==> Writing bsc/LICENSE
==> Writing bsc/README.md
```

如输出所示，*rebar3* 为我们的发行包生成了许多目录和文件，其中包括 *apps/bsc/src* 目录下的骨架式源文件，*config* 目录下的 *sys.config* 文件和 *rebar.config* 文件。后者包含的指令为 *rebar3* 提供了项目相关的一些细节，例如编译器标志、发布信息和依赖关系等。以下是 *rebar3* 为我们的 *bsc* 发布项目生成的基本的 *rebar.config*：

```
$ cd bsc
$ cat rebar.config
{erl_opts, [debug_info]}.
{deps, []}.

{relx, [{release, {'bsc', "0.1.0"},
                  ['bsc',
                   sasl]},

        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"},

        {dev_mode, true},
        {include_erts, false},
```



```

        {extended_start_script, true}}
    }.

    {profiles, [{prod, [{relx, [{dev_mode, false},
                                {include_erts, true}]}]}
                ]}
    }.

```

此 *rebar.config* 文件包含 4 个元组，这 4 个元组将在下面进行介绍。你可以修改这些设置中的任何一项或根据项目的需要添加其他设置。

- **erl_opts** 元组为 *erlc* 编译器提供了编译器选项。
- **deps** 元组声明了项目的依赖关系。幸运的是，*bsc* 除了标准的 Erlang/OTP 之外什么都不需要。
- **relx** 元组提供了与发行包生成有关的设置。*rebar3* 使用 *relx* 工具生成发行包。由于我们在本小节中的目标是使用 *rebar3* 生成 *bsc* 发行包，因此我们稍后会详细研究这些设置。
- **profiles** 元组提供了一种机制，使得我们可以根据不同的开发任务或角色设定不同的设置。顾名思义，此处生成的配置文件中的 **prod** 用于针对生成面向生产环境的发行包提供设置。

306

在生成的源文件骨架中，请特别注意 *application* 资源文件骨架，即 *apps/bsc/src/bsc.app.src*。*rebar3* 之所以生成此文件而非直接创建最终的 *application* 资源文件，是因为其将会在稍后作为编译过程的一部分，以 *bsc.app.src* 为蓝本，自动将所有 *application* 源代码模块的名称填入其 *modules* 定义中，然后再生成 *bsc.app* *application* 资源文件。当我们将 *bsc.app.src* 文件和 *rebar.config* 文件中生成的 "0.1.0" 版本号更改为正确的 "1.0" 以匹配 *bsc* 版本后，再执行编译便能理解这一点（可以使用任何文本过滤工具来实现此修改，此处我们采用的方法是使用一行 Perl 代码）：

```

$ perl -i -pe 's/0\.1\.0/1.0/' ./apps/bsc/src/bsc.app.src ./rebar.config
$ rebar3 compile
==> Verifying dependencies...
==> Compiling bsc

```

然后查看编译进程生成的 *_build/default/lib/bsc/ebin/bsc.app* 文件：

```

$ cat _build/default/lib/bsc/ebin/bsc.app
{application,bsc,
  [{description,"Base Station Controller"},
   {vsn,"1.0"}],

```

```
{registered,[]},
{mod,{bsc_app,[]}},
{applications,[kernel,stdlib]},
{env,[]},
{modules,[bsc_app,bsc_sup]},
{contributors,[]},
{licenses,[]},
{links,[]}}.
```

正如文件内容所示, *rebar3* 根据 *src* 目录中存在的 Erlang 模块自动为我们填写好了 *modules* 定义。当我们添加更多模块时, *rebar3* 会在编译阶段自动将它们添加到 *application* 资源文件中, 这比自己手动编辑资源文件要容易得多。唯一棘手的部分是, 如果你要修改 *application* 资源文件的其他字段, 则必须记住, 应当编辑的是 *bsc.app.src* 文件, 而不是生成的 *bsc.app* 文件。

307 为了运行此骨架 *application*, 我们可以启动一个 *rebar3* shell, 它可以确保所有项目路径都自动置于 Erlang 加载路径中。并且当 shell 启动时, 也启动了我们的 *application* :

```
$ rebar3 shell
==> Verifying dependencies...
==> Compiling bsc
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:0] [kernel-poll:false]

==> Booted bsc
==> Booted sasl

...<snip>...

==PROGRESS REPORT==== 26-Dec-2015::21:58:36 ===
      application: sasl
      started_at: nonode@nohost
Eshell V7.2 (abort with ^G)
1> application:which_applications().
[{sasl,"SASL CXC 138 11","2.6.1"},
 {bsc,"Base Station Controller","1.0"},
 {inets,"INETC CXC 138 49","6.1"},
 {ssl,"Erlang/OTP SSL application","7.2"},
 {public_key,"Public key infrastructure","1.1"},
 {asn1,"The Erlang ASN1 compiler version 4.0.1","4.0.1"},
 {crypto,"CRYPTO","3.6.2"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]
```


我们生成的骨架 `application` 中并不包含实际的代码，但它仍然能够启动并正确运行。请注意，`rebar3` shell 会启动一些 `bsc` 并不需要的其他 `application`，例如 `inets` 和 `ssl`。如果改为手动启动我们的 `application`，则不会出现它们。

为了把代码填入我们创建好的项目，可以找到 `bsc` 示例中的代码并复制过来，这些代码可以在本书的 GitHub 仓库中找到，位于对应本章的目录下：

```
$ cp -v <path-to-bsc-example-dir>/src/*.erl apps/bsc/src
<path-to-bsc-example-dir>/src/bsc.erl -> apps/bsc/src/bsc.erl
...
```

完成后，我们又可以使用 `rebar3` 来清理和编译项目了：

```
$ rebar3 do clean, compile
==> Cleaning out bsc...
==> Verifying dependencies...
==> Compiling bsc
```

如果再次启动 `rebar3` shell，可以看到这些 `application` 就如我们预期的那样运行：

```
$ rebar3 shell
...<snip>...
1> application:which_applications().
[{saspl,"SASL CXC 138 11","2.6.1"},
 {bsc,"Base Station Controller","1.0"},
 {inets,"INETC CXC 138 49","6.1"},
 {ssl,"Erlang/OTP SSL application","7.2"},
 {public_key,"Public key infrastructure","1.1"},
 {asn1,"The Erlang ASN1 compiler version 4.0.1","4.0.1"},
 {crypto,"CRYPTO","3.6.2"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]
```

308

使用 `rebar3` 创建发行包

`rebar3` 工具采用 `relx` 而非标准 Erlang/OTP 自带的 `reltool`，是为了降低开发人员创建发行包的难度，因为许多人反映正确配置和使用 `reltool` 太难了。

用 `rebar3` 创建一个发行包非常直截了当：

```
$ rebar3 release
==> Verifying dependencies...
==> Compiling bsc
==> Starting relx build process ...
```



```

====> Resolving OTP Applications from directories:
      /Users/francescoc/ernie2/bsc/_build/default/lib
      /Users/francescoc/ernie2/bsc/apps
      /usr/local/lib/erlang/lib
====> Resolved bsc-1.0
====> Dev mode enabled, release will be symlinked
====> release successfully created!

```

一旦生成了发行包，立刻就可以验证它是否能按预期工作：

```

$ _build/default/rel/bsc/bin/bsc console
Exec: /usr/local/lib/erlang/erts-7.2/bin/erlexec -boot ...
Root: /Users/francescoc/ernie2/bsc/_build/default/rel/bsc
/Users/francescoc/ernie2/bsc/_build/default/rel/bsc
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:30] [kernel-poll:true]

```

```

==PROGRESS REPORT==== 27-Dec-2015::11:37:56 ===
      supervisor: {local,sasl_safe_sup}
      started: [{pid,<0.49.0>},
                {id,alarm_handler},
                {mfargs,{alarm_handler,start_link,[]}},
                {restart_type,permanent},
                {shutdown,2000},
                {child_type,worker}]

```

...<snip>...

```

==PROGRESS REPORT==== 27-Dec-2015::11:37:56 ===
      application: sasl
      started_at: bsc@francescoc
Eshell V7.2 (abort with ^G)
(bsc@francescoc)1> application:which_applications().
[{sasl,"SASL CXC 138 11","2.6.1"},
 {bsc,"Base Station Controller","1.0"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]

```

除了像示例中那样向 `_build/default/rel/bsc/bin/bsc` 传递 `console` 参数（作用是启动 application 并为我们提供一个 Erlang shell），还可以改为传递 **start** 参数，这样将会在后台启动发行包；或者传递 **attach** 参数以附加到已启动的发行包的 shell；或者传递 **stop** 参数停止已启动的发行包。如果不传递任何参数直接执行 `_build/default/rel/bsc/bin/bsc` 则可以查看所有支持的参数和选项列表。



受 *rebar.config* 中 *relx* 元组针对默认发行包的设置参数的影响，此方式生成的发行包是用于开发环境而非生产环境的。该默认配置中设置了 *dev_mode* 为 *true*，这意味着用于创建发行包的 *application* 源文件实际上是一些指向 *apps/bsc/src* 目录下文件的链接。*dev_mode* 设置还将 *include_erts* 设置为 *false*，这将使 Erlang 运行时不会包含在生成的发行包中。如此种种设置对于开发而言非常方便，因为这使得开发人员可以在 *apps* 目录下编辑源文件，然后便可用于发行包构建，或者执行重编译并重加载使得直接在已运行的发行包中生效。除此之外，这些设置还使得开发人员可以直接使用系统中已安装的 Erlang，而不需要在每个发行包中都构建一个，这样一来便可以基于多种不同版本的运行时快速测试发行包。

幸运的是，要构建面向生产环境的发行包很容易，而且我们并不需要修改这些默认设置，这多亏了 *rebar3* 提供了 *profile* 机制。*rebar.config* 中的 *profiles* 元组中有一个 *profile* 的名字为 *prod*，其中将 *dev_mode* 设置为了 *false*，并将 *include_erts* 设置为了 *true*。要使用 *prod* 这一 *profile*，我们只需在命令行上使用 *rebar3* 的 **as** 指令即可：

```
$ rebar3 as prod release
==> Verifying dependencies...
==> Compiling bsc
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    /Users/francescoc/ernie2/bsc/_build/default/lib
    /Users/francescoc/ernie2/bsc/apps
    /usr/local/lib/erlang/lib
==> Resolved bsc-1.0
==> Including Erts from /usr/local/lib/erlang
==> release successfully created!
```

as 指令指示 *rebar3* 使用指定的 *profile* 执行命令。请特别注意本示例底部附近的粗体文本，它表明 *relx* 把 Erlang 运行时系统也一起打包了，这是遵照 *prod profile* 指示的结果。并且由于 *prod profile* 中将 *dev_mode* 设置为 *false*，所以如果你查看 *_build/prod/rel/bsc/lib/bsc-1.0/src* 目录，会发现源文件是被复制到发行包中的，而不是像制作默认发行包时那样是链接回 *apps* 源目录的。

而使用 *rebar3 tar* 指令则使创建发行包并打包为 *tar* 文件变得很简单：

```
$ rebar3 as prod tar
==> Verifying dependencies...
==> Compiling bsc
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    /Users/francescoc/ernie2/bsc/_build/prod/lib
```



```

/Users/francescoc/ernie2/bsc/apps
/usr/local/lib/erlang/lib
/Users/francescoc/ernie2/bsc/_build/prod/rel
====> Resolved bsc-1.0
====> Including Erts from /usr/local/lib/erlang
====> release successfully created!
====> Starting relx build process ...
====> Resolving OTP Applications from directories:
    /Users/francescoc/ernie2/bsc/_build/prod/lib
    /Users/francescoc/ernie2/bsc/apps
    /usr/local/lib/erlang/lib
    /Users/francescoc/ernie2/bsc/_build/prod/rel
====> Resolved bsc-1.0
====> tarball /Users/francescoc/ernie2/bsc/_build/prod/rel/bsc/bsc-1.0.tar.gz
    successfully created!

```

使用 rebar3 处理制作发行包时的项目依赖问题

到目前为止，我们的 *rebar3* 示例中仅包含了 *bsc* 一个 application，而且它不依赖于外部项目，但一般而言，许多 Erlang application 都会依赖于其他 application。幸运的是，*rebar3* 能够自动下载所依赖的其他项目，并将它们一起编译。

假设我们希望日志文件可以与某些现有的日志轮换工具协同运行，并且保护我们的 application 避免其在内存不足时由于大量进程出现持续性的意外错误而引发日志风暴，于是决定使用流行的开源框架 *lager* 来替换 *bsc* 中现有的日志记录机制。那么我们需要将 *lager* 的依赖关系添加到 *bsc* application 中，而这是非常简单的——我们只需在 *rebar.config* 文件的 *deps* 元组中指定此依赖即可：

```
{deps, [{lager, {git, "git://github.com/basho/lager.git",
                    {tag, "3.0.2"}}}]}
```

该指令告诉 *rebar3*，*lager* 是一个源依赖项，并且 *git* 元组告诉 *rebar3* 可以从哪个位置获取 *lager* 源代码，而 *tag* 元组指明了 *bsc* application 依赖于哪个版本的 *lager*。

这个指令写好后，可以试问 *rebar3* 当前的依赖关系是什么：

```

311 $ rebar3 deps
lager* (git source)

```

要求 *rebar3* 执行编译会使其自动获取 *lager* 依赖项的源代码以及 *lager* 自身所依赖的所有其他项目源代码：

```

$ rebar3 compile
====> Verifying dependencies...

```




```

====> Fetching lager ({git,"git://github.com/basho/lager.git",
                        {tag,"3.0.2"}})
====> Fetching goldrush ({git,"git://github.com/DeadZen/goldrush.git",
                        {tag,"0.1.7"}})
====> Compiling goldrush
====> Compiling lager
====> Compiling bsc

```

此编译是基于 default profile 的，所以如果在编译完成后查看 `_build/default/lib` 目录，我们将看到 `bsc`、`lager` 目录，并且还会看到 `goldrush` 目录——它正是 `lager` 所依赖的另一个项目：

```

$ ls _build/default/lib
bsc  goldrush  lager

```

为了构建带有 `lager` 的发行包，我们首先需要修改 `apps/bsc/src/bsc.app.src` 以将 `lager` 添加到 applications 列表中，位于 `kernel` 和 `stdlib` 之后。完成这些更改后，可以基于 default profile 创建一个发行包：

```

$ rebar3 release
====> Verifying dependencies...
====> Compiling bsc
====> Starting relx build process ...
====> Resolving OTP Applications from directories:
      /Users/francescoc/ernie2/bsc/_build/default/lib
      /Users/francescoc/ernie2/bsc/apps
      /usr/local/lib/erlang/lib
      /Users/francescoc/ernie2/bsc/_build/default/rel
====> Resolved bsc-1.0
====> Dev mode enabled, release will be symlinked
====> release successfully created!

```

如果查看 `_build/default/rel/bsc/lib` 目录的内容，可以看到 `rebar3` 构建了发行包中所有必需的 application：

```

$ ls _build/default/rel/bsc/lib
bsc-1.0  goldrush-0.1.7  lager-3.0.2

```

然后运行 application，并如我们所期望看到的那样，所有 application 确实正在运行：

```

$ _build/default/rel/bsc/bin/bsc console
...<snip>...
(bsc@francescoc)1> application:which_applications().
[{sasl,"SASL CXC 138 11","2.6.1"},

```



```
{bsc, "Base Station Controller", "1.0"},
{lager, "Erlang logging framework", "3.0.2"},
{goldrush, "Erlang event stream processor", "0.1.7"},
{compiler, "ERTS CXC 138 10", "6.0.2"},
{syntax_tools, "Syntax tools", "1.7"},
{stdlib, "ERTS CXC 138 10", "2.7"},
{kernel, "ERTS CXC 138 10", "4.1.1"}]
```

不仅 *bsc*、*lager* 和 *goldrush* 正在运行，而且还启动了标准的 *compiler* 和 *syntax_tools* application，因为 *goldrush* 用到了它们，你可以查看 `_build/default/lib/goldrush/src/goldrush.app.src` 文件中的 applications 列表部分，可看到其中列出了这些 application。

rebar3 提供的功能很丰富，比我们的 *bsc* application 所需的更多。它有一个可扩展和可定制的插件系统，与 Erlang 的 *common-test*、*dialyzer* 和 *eunit* 工具紧密配合，能够支持测试以及代码覆盖率计算和分析，并支持将发行包发布到 Erlang/Elixir 的 *hex* 包管理系统中。并且我们接下来还要在第 12 章中介绍 *rebar3* 还支持发行包的升级。

总结

想必你会同意，我们在这里展示的内容十分丰富，可能比你在阅读本章时最初希望看到的要更详细！话虽如此，将 OTP 应用捆绑在一个发行包中并将其作为一个单元启动的步骤事实上并不算多，而且相对比较简单。我们深入细节的原因是，不仅要告诉你如何做，而且希望你明白为什么那样做。在未来的某一天，当你需要将 Erlang/OTP 发行包集成到你的构建系统中时，或者需要排查一个多年正常运转的节点为何突然拒绝启动的原因时，你会感谢我们。说出来你可能难以相信，我们见过太多直接使用 `erl -s Module` 启动的系统了，这些系统每天、每小时甚至每分钟（并且在很多情况下甚至是几秒钟）需要处理数万次交易，但它们的启动方式是如此简陋，不符合 OTP 理念，没有配置 *heart*，没有设定为嵌入式目标系统以守护程序方式运行。正确的做法是，先创建一个适当的 OTP 发行包，并将此过程集成到构建系统中，这样才能构建一个好的基础，我们希望你总是以这样的方式去实践。

如果你的 Erlang 节点必须 7×24 无间断长年运行，那么最好将其以嵌入式目标系统的方式启动。你必须严格控制发行包中模块、application 和配置文件的版本。并且为了能在单台主机上运行多个嵌入式节点，访问 Erlang shell 时的 I/O 流应被设定为发送到 Erlang 根目录，而不是 `/tmp` 目录。

以守护进程方式启动你的 Erlang 系统，这样每次重新启动计算机或镜像时它都能自动启动。始终确保你随时可用 `erl` 命令，并有对应的引导文件，用于启动 *kernel*、*stdlib* 和



`sasl`，这样当节点崩溃并拒绝启动时，你可以查看本机上的 SASL 日志。不要忘记设置模拟器标志、一般标志和素参数，以满足内部的运行需要。你是否希望在使用 `+Bc` 禁用中断处理程序的同时仍然允许用户杀死 shell？是否希望打印出使用 `-emu_args` 传递给模拟器的那些参数，而且还希望使用 `-init_debug` 标志打印出启动跟踪报告？在实现和配置 `heart` 脚本来处理模拟器崩溃方面你又有何考虑？因为各种各样的组合实在太多，因此为你和你的组织确定最为恰当的配置可能需要多年的运营和应急处理经验才行。虽然有些困难，但你最终会具备那样的能力，希望你好好掌握我们在本章中提到的所有内容，那么同事对你的呼救将会少很多次，而且绝不会在半夜。

话虽如此，我们知道并非所有的系统都是在处理关键业务，都需要这样高级别的监督，都具备如此的复杂性和专业性。如果简单目标系统能够满足你的要求，那么它们同时是可以接受并值得重视的方案。如果在单台机器上共享同一份 Erlang 安装用于运行多个节点足以满足你的需要，则无须再为每个发行包单独包含自己的 Erlang 虚拟机。这样做的代价是你将无法单独升级 application 和模拟器，但是你可能并不需要！选择什么类型的发行包、什么部署方式最适合你和你的组织，最终还要由你判断，它可以按你的需要变得简单或复杂。重点是，你必须明白你所做的选择背后的取舍，而非投机取巧，否则你将最终在未来付出代价。

我们在本章中介绍的基于某些库和工具实现的自动化过程，包括以下步骤：

1. 为你的节点创建一个 `rel` 资源文件，在其中定义哪些部分将会被包含到你的发行包中。

该 `rel` 文件将列出所有 application 及其各自的版本号，以及要在目标部署中使用的模拟器版本号。

2. 创建一个引导文件，其中包含启动你的节点所需的所有命令。

3. 创建好文件 / 目录结构，它们将会部署到你的目标系统中。

其中包含 `lib`、`releases` 和 `bin` 目录，并且如果你打算把模拟器也附带，那么还将包含 `erts` 目录。

4. 针对你的部署需要（可能还需要考虑目标主机的情况），配置你的启动脚本。

包括 `start_erl.data` 文件和一些与部署需求紧密相关的配置文件，另外还有针对目标环境的配置脚本。

你可以在第 12 章的“为发行包创建升级”一节中找到这些步骤的其他示例，我们在那里创建了一个发行包，用的是第 6 章中介绍的咖啡机 FSM 示例，并为其做软件升级。但是，



314 如果你懒得每次都重复做这些杂事（我们也是），而且不需要集成到现有的构建和发布基础设施中，那你可以使用现有的工具和库来完成大部分工作，并自动完成其余工作，使用 *rebar3* 可以大大简化这一切。

至此，你已在本书中遇到了许多种不同类型的文件，它们共同构成了发行包。我们将它们列在表 11-1 中，现在正是回顾它们的好机会。

表 11-1: Erlang/OTP 文件类型

文件类型	文件扩展名	描述
Erlang 模块	<i>.erl</i>	包含 Erlang 源代码的文件
已编译模块	<i>.beam</i>	Erlang 源代码文件编译后的文件，可由 BEAM 模拟器执行
application 资源文件	<i>.app</i>	包含 application 资源和配置数据的文件
application 升级文件	<i>appup</i>	包含 application 升级数据的文件
发行包文件	<i>.rel</i>	包含与发行包有关的 application 和模拟器版本号的文件
发行包升级文件	<i>relup</i>	包含发行包升级信息的文件
启动脚本	<i>.script</i>	文本版本的脚本文件，用于引导系统启动
二进制启动脚本	<i>.boot</i>	二进制版本的脚本文件，用于引导系统启动
配置文件	<i>.config</i>	包含与 application 相关的环境变量的文件

我们在第 12 章将介绍 *.appup* 和 *relup* 文件，它们用于 application 的实时升级以及模拟器的定期升级。

如果你还觉得讲解的不够并且想要了解更多关于创建发行包的信息，请直接阅读 Erlang/OTP 附带的文档。《OTP 设计原则用户指南》(*The OTP Design Principles User's Guide*) 将告诉你更多关于发行包和处理发行包的信息，并引导你创建第一个发布包，以能够在目标环境中部署。《OTP 系统原则用户指南》(*The OTP System Principles User's Guide*) 中有一些章节涵盖了系统的启动、重启和停止等方面，并更详细地描述了嵌入式与交互式两种模式下，代码加载策略的差异。它的内容与《OTP 设计原则用户指南》中的内容有重叠，因为其中也有涉及目标系统的创建和配置方面的内容。并且，用户指南中还介绍了 *target_system.erl* 模块，该模块来自 *sasl* application 的 *examples* 目录，本书的 GitHub 代码仓库中对应本章节的目录中也有该模块。它是一个示例，在解释如何构建一个发行包和目标系统的同时，将我们在本章讲到的许多步骤都自动化了，在 *rebar*、*rebar3*、*relx* 和 *reltool* 这些工具出现之前确实需要遵循这些步骤。请看看它，因为它为那些将 Erlang 整合到现有的构建系统中的人们提供了很多灵感。

参考手册页是用户指南的进一步补充，其中的以下内容与我们刚刚介绍的内容有关，因



此值得一提。

- 如果你需要了解关于 `rel` 文件的更多信息，请查阅 `rel` 参考手册页。给定一个 `rel` 文件，`systools` 提供了一系列函数，可帮助创建启动脚本、引导文件和目标 `tar` 文件等。二进制引导文件及其脚本文件对应的内容在 `script` 参考手册页面中有更详细的描述。要详细了解它们的执行方式，请查看 `init` 用户手册页。
- 有时你可能需要在目标机器上自动执行任务，并将发行包制作过程与你使用的其他工具（例如用于系统的非 Erlang 部分的某些工具）相集成。如果是这种情况，请阅读 `release_handler` 手册页，其中介绍了一些函数，允许你解压和安装由 `systools` 调用创建的 `tar` 文件。但是，它假定了目标主机上已经安装过一个 Erlang 并运行，而这一点并非总是能满足。我们在第 12 章讨论实时升级时会更详细地介绍这个库。
- 如果你需要远程加载代码，并且本章中的示例不够用，那么 `erl_boot_server`、`erl_prim_loader` 和 `init` 用户手册页将会对你有所帮助。
- `erl` 和 `init` 手册页描述了很多虚拟机标志和命令行标志，其中一些我们在本章中未涉及。对于各个参数，你必须参考使用这些参数的模块和 `application` 的用户手册页。
- `heart` 手册页是查找有关自动启动相关信息的地方，包括配置细节和实现脚本时需要用到的各种环境变量等。你会看到一些在 `erl` 手册页中描述的环境变量。
- 如果你是在 Windows 中运行，请阅读 `start_erl` 手册页。它的功能等同于我们在本章中使用的 `start` 命令，但允许你在 Windows 环境下启动嵌入式系统。

我们没有介绍 `reltool` 工具，但它有用户指南和参考手册页，如果你的系统需要复杂的配置，超出了 `rebar3` 所能处理的范畴，你可能会发现它很有用，尽管这种情况非常罕见。`rebar3` 的信息可以在 <https://www.rebar3.org> 上找到，而 `relx`，你可以使用 `rebar3` 或从 GitHub 获取 (<https://www.rebar3.org/>)。

如果所有这些看起来都很吓人，那么直接简单地使用 `rebar3` 就好。它可以为各种类型的项目构建和创建发行包，如果有特殊需求可以通过其插件系统进行扩展，还可以下载和帮助管理其他项目的依赖项，并且可以与 `hex` 包管理系统一起使用，使你能发布你的系统，以便 Erlang 社区中的其他人可以使用它。有关 `rebar3` 和 `hex` 的更多信息，请参阅 `rebar3` 文档 (<https://www.rebar3.org/docs/hex-package-management>)。



接下来是什么

Erlang 不仅是一种适用于解决特定类型问题的语言，更超越了语言的范畴，它提供了一整套工具，用于开发、部署和监视系统，助你打造出可预期、可维护的系统。在本章中，我们介绍了如何打包和部署你的第一个目标系统，但这只是你冒险旅程的开始。接下来我们将介绍如何通过实时升级来管理错误修复和部署新功能。我们将介绍一些与升级有关的工具和功能，它们是内置于 OTP 及各个 behavior 中的。你已听说过 Erlang 达到了 5 个 9 级别的可用性吧，但你知道这甚至是将软件维护和升级时间也算进去的吗？继续阅读你会了解 Erlang 是如何做到这一点的。



发行包升级

在你的系统上线后，它日复一日地在后台马不停蹄地处理请求。它能够在出现问题时自行修复，并在断电或系统关机重新引导后自动重新启动。但是与任何一款软件一样，你得继续优化它，修复发现的错误并添加新功能。你可能需要面对各种场景，例如你的咖啡机可能会有数千个实例在同时运行，它们每一个都运行在专用的硬件平台上，并且可通过无线链路监控，或者你需要设计一个要求保障 100% 可用性的系统，而且连升级时间也计算在内，无论是哪一种情况，都迫使你不得不认真研究和掌握 Erlang/OTP 的软件升级功能。想象一下，某个早晨，由于办公室的咖啡机正在执行固件升级，导致你无法享用咖啡时你的心情会有多糟！

Erlang VM 内置的模块动态加载能力可应对简单的向后兼容的修补工作。但你是否考虑过涉及更改函数 API 时的情况？或者，由于协议发生变化，导致运行旧版代码的进程无法与运行新版代码的进程相互通信的情况？当发行包版本变化或数据库模式变化后，如何随之处理进程内部的循环数据状态更新？更重要的是，如果升级出现故障你需要降级又会怎么样？

复杂系统需要以协调和可控的方式升级。而 Erlang/OTP 提供的良好的基础设施，包括内置的模块动态加载功能在内，为构建协调和控制这些升级的工具奠定了基础，大大减少和隐藏了该项任务的复杂性。在介绍这些工具之前，让我们重温一下与我们的例子有关的语义、术语和一些常用函数，以确保我们对它们有一致的认识。

软件升级

我们在第 2 章的“模块升级”一节曾介绍过模块级别的升级。如果你已经阅读过，可能还记得，要想在 Erlang 运行时环境中加载新模块，可以使用 shell 命令 `l(Module)` 或调用 `code:load_file(Module)`，而如果你仅想编译源代码，则可以使用 `c(Module)` 或

`make:files(ModuleList,[load])` 等。在任一时刻, Erlang 运行时环境针对任何一个已加载的模块都可以持有两个版本的代码。我们分别将它们称为 *old* (旧) 和 *current* (当前) 版本。运行旧版本模块的进程将继续执行, 直到它发出全限定式函数调用——即格式为 `Module:Function(...)` 的调用, 其中模块名称被用作该函数的前缀。

当发生全限定式函数调用时, 运行时检查以确保进程正在运行的是 *current* 版本的代码。如果是, 调用将继续使用当前的代码。但是, 如果发现进程仍在使用 *old* 版本的代码, 则在调用之前, 指向代码的指针将切换到 *current* 版本。

调用库模块必须是全限定式的, 因为你正调用的是另一个模块, 所以这样的调用将自动使用 *current* 版本。但是, 用于控制进程接收 - 求值 (receive-evaluate) 循环的递归调用, 往往是本地式的而非全限定式的。我们需要将这些本地式调用改为全限定式的, 或者可以在接收 - 求值循环中添加一条用于触发全限定式函数调用的新消息。根据升级的复杂性不同, 此函数可以调用新模块中的循环函数, 也可以在返回进入循环之前调用新模块中负责处理进程状态 (包括循环数据、ETS 表和数据库模式) 改变的钩子函数。

运行 *current* 版本的模块的进程, 只要一直不执行全限定调用, 即使系统加载了新版本模块后, 模块也并不会随之切换但仍能够继续运行。但该进程所运行的模块版本将被视为 *old* 版本 (而不再是 *current* 版本), 那么之后当一个比 *current* 更新版本的模块被加载时, 该进程将被无条件终止。此外, 如果使用 `code:purge(Module)` 调用强制删除 *old* 版本的模块, 也会迫使那些运行此版本的进程无条件终止。

双模块版本限制

当初为了简化 JAM 虚拟机 (曾是当时使用最多的虚拟机) 的设计, 并使其能够在小内存环境下节约资源, 于是设立了同一个模块只能并存两个版本的限制 (即双模块版本限制), 如今已变成一个历史遗留问题。如今, 正确的设计决策应是在运行时允许不限数量的模块版本, 并在不再使用时进行垃圾回收。但是在 JAM 中, 要想对代码做垃圾回收, 你必须遍历每一个进程的堆栈并获取其每一个函数调用的返回地址, 由此确定进程正在使用哪个版本的模块。这样的操作非常耗时, 开发人员倾向于避免, 所以他们决定通过双模块版本限制来简化它。

由于运行时系统中支持双版本的代码, 所以我们需要有一种方案能够确定模块的当前版本号。-`vs`n(Version). 模块属性可帮助我们实现这一点。Version 可以是任何 Erlang 数据项, 但它通常是一个字符串、数字或原子。通常情况下, 它是由版本控制系统在提交代码至存储库时自动触发的脚本所设置的 (例如, 如果使用 Git 进行源代码管理, 则可以将 Version 设置为 `git describe --long` 命令所输出的字符串中的内容, 该命令的作用是获取最新的一个 Git 标签, 以及自该标签以来的提交数和当前提交的散列)。

将 `vs_n` 属性以及其他相关属性置于模块开头处，使我们能够确定要升级的代码的版本，此版本值有助于我们控制进程状态、数据库模式、协议以及其他非向后兼容的内部数据格式的变更。你可以调用 `Mod:module_info/0,1` 获取当前模块的版本值。

`vs_n` 属性不是必需的。如果省略，编译器会在编译时使用 `beam_lib:md5/1` 调用生成模块的 128 位 md5 摘要值。计算此 md5 摘要会涉及模块的一些属性，但不包括编译日期和与代码无关的其他属性，因为它们可能会在代码本身不变的情况下发生变化。这种策略保证了无论编译时间、空格、回车或代码中的注释如何变化，版本号都将被计算为相同的 128 位摘要值。

还记得我们在第 6 章“Coffee FSM”一节中看过的 FSM 示例吗？让我们掸掸灰把它拿出来编译一下，以更好地理解 `vs_n` 模块属性的工作原理。如果你使用本书 GitHub 代码库中的模块，它位于 `ch12/erlang/coffee.erl.original`。不要忘记将其文件名更改为 `coffee.erl`。然后你可以编译它，如下所示：

```
1> c(coffee).
{ok,coffee}
2> coffee:module_info(attributes).
[{vs_n,[293551046745957884913825426256179654413]}]
3> {ok, {coffee, MD5Digest}} = beam_lib:md5(coffee).
{ok,{coffee,<<220,215,224,7,110,247,231,148,86,224,44,
74,197,2,111,13>>}}
4> <<Int:128/integer>> = MD5Digest, Int.
293551046745957884913825426256179654413
```

在 shell 命令 2 中，我们调用 `coffee:module_info/1` 返回了 `vs_n` 模块属性中的 md5 摘要值，然后在 shell 命令 3 和 4 中从模块本身直接计算摘要，并确定了得出的值与前者完全相同，从而逆向明确了这一过程。现在让我们在模块中手动添加 `-vs_n` 指令并重新编译：

```
-module(coffee).
-export(...).

-vsn(1.0).

...
```

这样一来，编译器就不再会用 md5 摘要值改写版本号了，而是按照我们的要求将其设置为 1.0：

```
5> c(coffee).
{ok,coffee}
```

```
6> coffee:module_info(attributes).  
[{vsn,[1.0]}]
```

接下来，让我们继续在此咖啡机 FSM 的基础上增加一种新的升级消息，用于触发全限定式函数调用。这将允许我们以受控的方式升级服务器，理解进程中涉及升级的所有步骤的具体做法和背后的原因。之后，我们将探讨如何使用 OTP 方式来完成。

第一个版本的咖啡机 FSM

也许你还记得，我们的 Erlang 版咖啡机 FSM 由三个状态组成，分别为 *selection*、*payment* 和 *remove*（参见图 12-1）。为了我们的软件升级示例，我们添加了一个名为 *service* 的新状态，该状态允许我们打开柜门并为咖啡机做维护。但在这么做之前，要添加一些通用的代码，用于执行全限定式调用，这为我们提供了一种基准，可以使用它来执行升级。我们既可以在每次接收 - 求值循环中都使用全限定式调用实现这一目的，也可以通过向进程发送消息触发全限定式调用的方式做到。

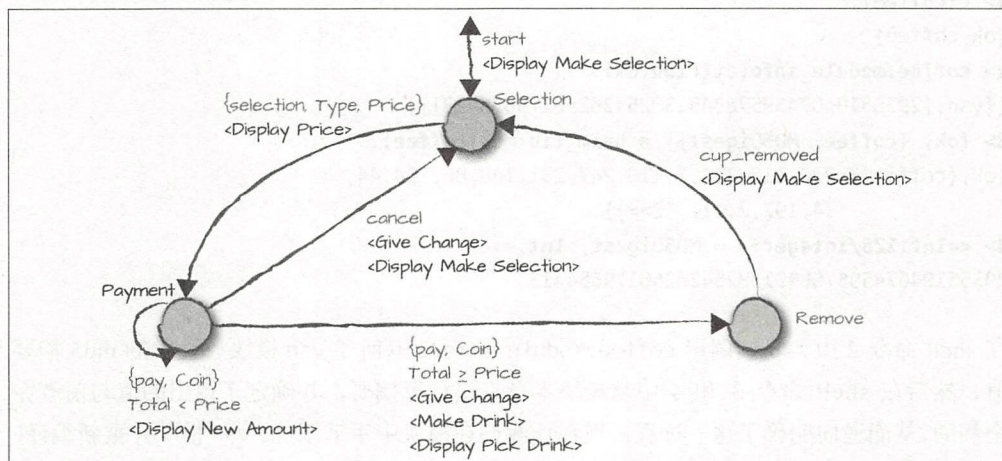


图 12-1: 咖啡机 FSM

321 升级代码时建议将加载新模块的方式与触发进程升级的过程分离。在我们的通用升级代码中，我们使用 `module:load_file/2` 加载模块。然后，通过发送 `{upgrade, Data}` 消息来通知必须升级的进程，它们将在内部发起全限定式调用来触发升级过程。

`Data` 是一种不透明的数据类型，其中包含新模块使用的与升级相关的信息。即使此刻没有用到它也应当保留作为占位用，因为未来的代码必将用到它，它的作用是让我们能够在转换到新模块的过程中操纵进程状态随之进行转换。举一个例子，假设我们正在升级频率服务器，并想增加更多的频率。在升级进程中，我们可以利用 `Data` 将新频率传递

给服务器。接收到该升级消息及此数据的进程向 code_change/2 发出完全限定的函数调用，其中第一个参数是进程状态，第二个参数是 Data。在这个函数中，我们可以将新频率添加到可用频率列表中，并使用新更新的循环数据在新模块中进入接收 - 求值循环。

我们来看看咖啡机 FSM 中的通用升级代码是什么样子的。请注意，我们已经为模块添加好了一个版本号：

```
-module(coffee).
-export(...).
-export([..., code_change/2]).
-vsn(1.0).
...

%% State: drink selection
selection() ->
    receive
        ...
        {upgrade, Data} ->
            ?MODULE:code_change(fun selection/0, Data);
        ...
    end.

%% State: payment
payment(Type, Price, Paid) ->
    receive
        ...
        {upgrade, Extra} ->
            ?MODULE:code_change({payment, Type,
                                Price, Paid}, Extra);
        ...
    end.

%% State: remove cup
remove() ->
    receive
        ...
        {upgrade, Data} ->
            ?MODULE:code_change(fun remove/0, Data);
        ...
    end.

code_change({payment, Type, Price, Paid}, _) ->
    payment(Type, Price, Paid);
```

```
code_change(State, _) ->
    State().
```

请注意，我们需要在所有状态中都处理 {upgrade, Extra} 消息。收到该消息后，我们对 code_change/2 发起全限定式的函数调用，其中第一个参数是 FSM 当前状态和循环数据，第二个参数是 Extra，将它们透明地传递给该调用。新模块中的 code_change/2 函数负责将旧进程状态更改为与新代码相兼容的新状态，在这一过程中可能会用到 Extra。进程状态的改变可能包括对循环数据格式和内容的修改、数据库模式的更改、与其他进程的同步、更改进程标志，甚至包括操纵邮箱中的邮件等。

一旦完成，code_change/2 以尾递归调用新的接收 - 求值循环的方式来退回控制权。在我们的示例中，这些函数是 FSM 状态函数 selection/0、payment/3 和 remove/0。这是该模块的第一个版本，所以不需要我们添加的 code_change/2 子句做任何实质性的事情；它们只是简单地回到发起调用前的状态即可。添加这些子句可避免我们尝试升级操作而进程正运行的是旧版本的 coffee 模块，于是出现“未定义函数”这类我们已解释过的运行时错误。

这是我们的基准代码。如果你使用本书代码库中的代码，则可以在对应本章的 erlang 目录中找到此代码。接下来让我们编译它，启动 Erlang 虚拟机，并让咖啡机 FSM 启动并运行，以确保它能够正常工作——在我们创建新版本模块并执行升级之前：

```
$ cd erlang
$ cp coffee.erl.1.0 coffee.erl
$ erl -make
Recompile: coffee
Recompile: hw
$ erl -pa patches
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> coffee:start_link().
```

```
Machine:Rebooted Hardware
```

```
Display:Make Your Selection
```

```
{ok,<0.36.0>}
```

```
2> coffee:module_info(attributes).
```

```
[{vsrn,[1.0]}]
```

```
3> coffee ! {upgrade, {}}.
```

```
{upgrade,{}}
```

```
4> coffee:module_info(attributes).
```

```
[{vsrn,[1.0]}]
```


请注意，在 shell 命令 3 中，我们是如何在未加载新版 FSM 的情况下触发升级的。这导致在当前版本的模块中执行了 `code_change/2` 调用。

添加一个新状态

添加一个 *service* 状态使得我们可以为咖啡机 FSM 做维护。当咖啡机 FSM 处于 *selection* 状态并打开柜门时将触发转入 *service* 状态。在任何其他状态下，开门事件都将被忽略。正如我们在图 12-2 中看到的那样，关闭柜门会触发硬件的重新启动并转换回 *selection* 状态。在所有其他状态下，关门事件都将被忽略。

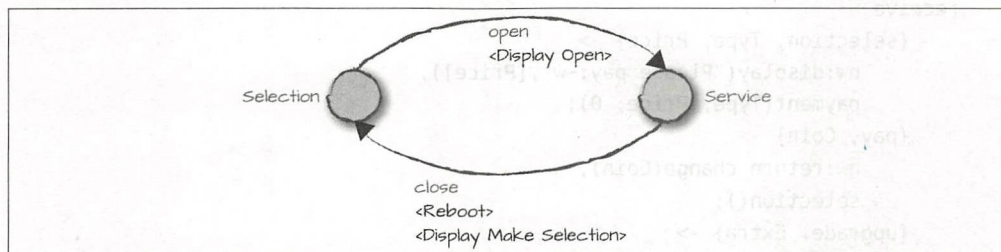


图 12-2: service 状态

现实中我们可以通过改写 *hw.erl* 以添加函数 `hw:lock()` 和 `hw:unlock()` 轻松地在硬件中增加锁控制功能，但我们为了保持示例简单而没有在此处这么做。增加锁控制将更有保障，确保咖啡机门只能在 *selection* 状态下打开，并在机器处于其他状态时保持锁定状态而禁止打开。

让我们来看看新版本的模块，在这里我们突出显示了其相较于 1.0 版本的变化。主要区别在于新增的 *service* 状态、*open* 和 *close* 事件以及在 `code_change/2` 函数子句中执行的操作有所不同。

首先，我们看到客户端函数 `open/0` 和 `close/0`，它们分别在咖啡机门打开和关闭时产生事件。在状态 *selection* 中，一旦收到 *open* 事件后，我们便显示出 *Open* 字样并转换到 *service* 状态。

service 状态忽略除用户插入硬币和关闭咖啡机门外的所有其他事件。一旦门关闭，硬件将重新启动，显示屏将指示客户进行选择。在除此之外的其他所有状态中，*open* 和 *close* 事件都将被忽略：

```
-module(coffee).  
-export([tea/0, espresso/0, americano/0, cappuccino/0,  
        pay/1, cup_removed/0, cancel/0, open/0, close/0]).  
-export([start_link/0, init/0, code_change/2]).
```

◀ 324

```

-vsn(1.1).

start_link() ->
    ...

open() -> ?MODULE ! open.
close() -> ?MODULE ! close.

...

selection() ->
    receive
        {selection, Type, Price} ->
            hw:display("Please pay:~w",[Price]),
            payment(Type, Price, 0);
        {pay, Coin} ->
            hw:return_change(Coin),
            selection();
        {upgrade, Extra} ->
            ?MODULE:code_change(fun selection/0, Extra);
    open ->
        hw:display("Open", []),
        service();
    _Other -> % cancel
        selection()
    end.

...

service() ->
    receive
        close ->
            hw:reboot(),
            hw:display("Make Your Selection", []),
            service();
        {pay, Coin} ->
            hw:return_change(Coin),
            service();
        _Other ->
            service()
    end.

...

code_change({payment, _Type, _Price, Paid}, _Extra) ->

```



```

hw:return_change(Paid),
hw:display("Make Your Selection", []),
selection();
code_change(State, _) ->
State().

```

在我们的 `code_change` 函数中，如果用户已经选择了饮料并正在为其付费，我们会返回已支付的全部金额并转换到 `selection` 状态。对于所有其他状态，则直接转回对应的升级前状态。在我们的例子中，实际上并未用到 `Extra`，但是考虑到我们正在准备的是代码在未来的升级，而我们此刻并不知道这些升级的具体细节，所以预留这个参数以确保支持未来的代码升级过程是值得的，并且这样做使我们可以将变量传递给未来的升级过程用于更改进程状态。

我们将 1.1 版本的源代码放在 `patches` 目录中并进行编译。请注意我们使用了 `-pa patches` 指令启动 Erlang 运行时系统。当第一次启动咖啡机 FSM 时，这个目录是空的。当我们发现并修复错误时，在这里放置新的 `beam` 文件。由于该目录出现在代码搜索路径列表中的顶部，因此放在这里的 `beam` 文件将覆盖代码搜索路径中稍后出现的相同模块的 `beam` 文件。在另一个 shell 中，输入：

```

$ cd erlang/patches/
$ erl -make
Recompile: coffee

```

在启动咖啡机 FSM 1.0 版本相同的 Erlang 节点上，通过调用 `code:load_file/1` 加载模块的新版本。代码服务器会在其代码搜索路径中查找 `coffee beam` 文件的第一个版本，而由于我们指定的 `patches` 目录位于列表顶部，因此选中的是刚编译的新版本。该操作的成功可通过 shell 命令 6 得到确认，因为其中显示的版本属性为 1.1：

```

5> l(coffee).
{module,coffee}
6> coffee:module_info(attributes).
[{vsn,[1.1]}]

```

此时，我们在运行时系统中加载了两个版本的 `coffee` 模块：刚加载的 `current` 版本和 FSM 进程使用的 `old` 版本。当我们在 shell 命令 7 中选择 `espresso` 并在随后的命令中购买选择的 `espresso` 时，因为我们采用的是全限定式的调用，因此 shell 会使用代码的 `current` 版本（即刚加载的版本）执行。然而，FSM 进程此时仍在 `coffee` 模块的 `old` 版本。

如果我们现在要加载另一个版本的咖啡机模块,即使是 1.0,咖啡机 FSM 进程也会被终止,因为它正运行着被删除了的旧版本的代码。current 版本将转而被视为 old 版本,而新加载的模块将成为新的 current 版本。在我们的例子中,没有这样做,但如果你编译了代码并且追随着我们的步骤,那么请自己尝试一下。

在 shell 命令 9 中,我们触发了升级。这导致当前处于 *payment* 状态的咖啡机 FSM 在新模块中调用 *code_change/2*。它退回了已投入的钱,并且,感谢新状态 *service*,现在我们能够打开和关闭机器门,以便维修了:

```
7> coffee:espresso().
```

```
Display:Please pay:150
```

```
{selection,espresso,150}
```

```
8> coffee:pay(100).
```

```
Display:Please pay:50
```

```
{pay,100}
```

```
9> coffee ! {upgrade, {}}.
```

```
Machine:Returned 100 in change
```

```
Display:Make Your Selection
```

```
{upgrade, {}}
```

```
10> coffee:open().
```

```
Display:Open
```

```
open
```

```
11> coffee:espresso().
```

```
{selection,espresso,150}
```

```
12> coffee:close().
```

```
Machine:Rebooted Hardware
```

```
Display:Make Your Selection
```

```
close
```

这正是 Erlang 处理升级的基本原理。通用代码部分是对 *{upgrade, Extra}* 消息的处理以及对 *code_change/2* 的调用,并且在 *code_change/2* 代码中采用全限定式调用返回到调用接收 - 求值循环。这在所有进程中都是相同的。而进程之间不同的地方则在于我们根据循环数据、进程状态和 Extra 内容不同而所做的不同的事。以这些知识为基础,接下来让我们继续阅读并了解如何使用 OTP 完成这些操作。

为发行包创建升级

为了使用 OTP 提供的工具和设计原则对发行包进行升级,我们必须以一个遵循第 11 章中介绍的原则正确打包和部署的 OTP 发行包为基础,并且还需要:

- 一个或多个已有 application 的新版本。

- 零个或多个新 application。
- 针对每个发生变更的 application 对应的 application 升级文件。
- 发行包资源文件与发行包升级文件。

包含错误修复和新功能的模块被打包到新的或现有的 application 中，其版本号被升高。 application 升级文件包含一些命令，告诉我们如何从一个 application 版本升级或降级到另一个 application 版本。发行包资源文件，我们曾在第 11 章的“发行包资源文件”一节中介绍过，它包含了构成新发行包的模拟器和 application 的版本清单。连同我们正在升级的基准系统的 application 升级文件和发行包文件，新发行包文件将用于生成发行包升级文件。该文件包含升级过程中必须执行的所有命令。在目标机器上安装新代码后，我们运行发行包升级文件中的指令。如果碰到任何错误，则使用旧发行包重新启动系统。你还需要通过测试和观察，确定系统是否稳定。如果稳定，就执行永久化。如果未做永久化而重启系统，系统将会基于旧发行包重启。让我们进行升级，看看不同的步骤和组件是如何协同工作的。

在本书的代码存储库中，对应本章节的目录中，你将找到我们曾用来创建第一个部署时所用的文件。我们采用了 *coffee_fsm.erl* 作为示例，并创建了一个 OTP application，其中包括对应 supervisor 和 behavior 的文件。我们还创建了 *coffee.app* 文件并将其放置在 *ebin* 目录中。请下载并编译这些代码，并确保你可以启动并运行它：

```
$ cd coffee-1.0/src ; erl -make ; mv *.beam ../ebin ; cd ../..
Recompile: coffee_app
Recompile: coffee_fsm
coffee_fsm.erl:2: Warning: undefined callback function
                        code_change/4 (behaviour 'gen_fsm')
coffee_fsm.erl:2: Warning: undefined callback function
                        handle_event/3 (behaviour 'gen_fsm')
coffee_fsm.erl:2: Warning: undefined callback function
                        handle_info/3 (behaviour 'gen_fsm')
coffee_fsm.erl:2: Warning: undefined callback function
                        handle_sync_event/4 (behaviour 'gen_fsm')
Recompile: coffee_sup
Recompile: hw
$ erl -pa coffee-1.0/ebin
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V7.2 (abort with ^G)
1> application:start(sasl), application:start(coffee).

...<snip>...
```

```
=PROGRESS REPORT===== 10-Jan-2016::21:27:28 ===
```

```
application: coffee
started_at: nonode@nohost
```

```
ok
```

```
2> coffee_fsm:module_info(attributes).
```

```
[[behaviour,[gen_fsm]],{vsn,['1.0']}]
```

328 尽管 *coffee* application 目录尚未位于 *lib* 目录中，但为了清晰起见，我们也给它定了一个版本号。请注意，在编译代码时，我们会得到以下警告：

```
Warning: undefined callback function
code_change/4 (behaviour 'gen_fsm')
```

在此之前，我们曾要求你耐心地忍受，忽略这个警告信息，但没有解释更多。但你现在应该已经了解它是什么了，并且知道之后升级 *coffee_fsm* 模块时我们会如何用到它。另外请注意，在 shell 命令 2 中获取模块属性时，我们同时获取到了 *behavior* 类型和当前模块的版本号。

运行我们的 application 后，让我们着手创建引导文件（boot file）、发行包文件（release file）和目标目录结构吧。我们在本书的代码库中使用的是空的 *sys.config* 和 *coffee-1.0.rel* 文件。如果你在阅读本文时也在同步实验，请自行启动并运行你自己的版本，不要忘记将 rel 文件中的标准 OTP application 和 *erts* 版本更新为你当前使用的 Erlang 版本。如果你没有在实验，或者无法访问代码，为了方便起见，我们在此已经包含了 *sys.config* 和 *coffee-1.0.rel* 文件的内容。如果是基于你自己使用的不同的 Erlang 版本做测试，则可能需要修改其中各个标准 OTP application 的版本号。

```
$ cat sys.config
```

```
[].
```

```
$ cat coffee-1.0.rel
```

```
{release,
  {"coffee","1.0"},
  {erts, "7.2"},
  [{kernel, "4.1.1"},
   {stdlib, "2.7"},
   {sasl, "2.6.1"},
   {coffee, "1.0"}]}.
```

```
$ mkdir ernie
```

```
$ erl
```

```
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]
```

```
Eshell V7.2 (abort with ^G)
```

```
1> systools:make_script("coffee-1.0", [{path, ["coffee-1.0/ebin"]}]).
```

```
ok
```



```

2> systools:make_tar("coffee-1.0",[{erts, "/usr/local/lib/erlang/"},
    {path, ["coffee-1.0/ebin"]},
    {outdir, "ernie"}]).
ok
3> halt().

$ cd ernie; tar xf coffee-1.0.tar.gz; rm coffee-1.0.tar.gz
$ mkdir bin; mkdir log
$ cp erts-7.2/bin/run_erl bin/.; cp erts-7.2/bin/to_erl bin/.
$ cp erts-7.2/bin/start.src bin/start
$ cp erts-7.2/bin/start_erl.src bin/start_erl
$ perl -i -pe "s#%FINAL_ROOTDIR%#%PWD#" bin/start
$ diff erts-7.2/bin/start.src bin/start
27c27,28
< ROOTDIR=%FINAL_ROOTDIR%
---
> ROOTDIR=/Users/francescoc/ernie
$ echo '7.2 1.0' > releases/start_erl.data

```

我们现在需要创建升级和降级版本所需的 *releases/RELEASES* 文件。在前一章中我们没有提到它，是因为只有在升级出故障需要降级回此版本发行包时才真正需要它。当我们进行升级并且该文件不存在时，会创建一个新文件，但仅包含此次升级信息。如果升级成功，那自然是好的，因为当我们第二次升级时，必须能够降级回第一次升级后的版本。但不足之处在于，如果第一次升级就出故障，我们将无法降级回最初的版本。一旦升级操作持久化了，我们将不得不重新安装节点。创建如下文件：

```

$ bin/start
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.1 (^D to exit)

1> application:which_applications().
[{coffee,[],"1.0"},
 {sasldb,"SASL CXC 138 11","2.6.1"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]
2> RootDir = code:root_dir().
"/Users/francescoc/ernie"
3> Releases = RootDir ++ "/releases".
"/Users/francescoc/ernie/releases"
4> RelFile = Releases ++ "/coffee-1.0.rel".
"/Users/francescoc/ernie/releases/coffee-1.0.rel"
5> release_handler:create_RELEASES(RootDir, Releases, RelFile, []).
ok

```

RELEASES 文件包含一个列表，其中每个已安装版本发行包对应一个条目。每个条目的信息都与 *rel* 文件中的信息相似，包括发行包和 *erts* 版本。然而，与 *application* 名称和版本一起，还包括 *application* 目录的绝对路径。尽管首个版本的 *RELEASES* 文件中仅包含单个条目，但后续升级将使其包含多个条目：

```
%% File:releases/RELEASES
[{release,"coffee","1.0","7.2",
  [{kernel,"4.1.1",
    "/Users/francescoc/ernie/lib/kernel-4.1.1"},
  {stdlib,"2.7",
    "/Users/francescoc/ernie/lib/stdlib-2.7"},
  {sasl,"2.6.1",
    "/Users/francescoc/ernie/lib/sasl-2.6.1"},
  {coffee,"1.0",
    "/Users/francescoc/ernie/lib/coffee-1.0"}],
  permanent}].
```

负责升级的代码

现在已经启动并运行了我们的第一个 OTP 兼容的发行包，接下来让我们创建新版本的 *coffee_fsm* 模块，添加新的 *service* 状态及其客户端函数。我们首先将版本属性提升为 1.1。它现在可能没有多大意义，但是如果你已经遵守了提升版本的规则（或者是通过脚本自动在为代码打标签或构建发行包时执行），你将在未来一次次的升级过程中得到回报，特别是在不久后的某个早上，当你试图弄清楚为什么你以为应当在生产中运行的代码版本实际上却并不是时。^{注1}

我们导出了状态函数 *service/2* 和 *service/3*（你可能还记得，*gen_fsm* 回调 *State/2* 用于处理异步事件，*State/3* 用于处理同步事件）。我们还导出了两个客户端函数，*open/0* 和 *close/0*，它们将咖啡机门的 *open*（开启）和 *close*（关闭）事件异步发送到 FSM。最后，我们导出 *code_change/4*，一个用于更新 *behavior* 状态的 *behavior* 回调。只要阅读过本章前面的“添加一个新状态”一节，这些内容你应该很熟悉：

```
-module(coffee_fsm).
-behavior(gen_fsm).
-vsn('1.1').
-export([start_link/0, init/1]).
-export([selection/2, payment/2, remove/2, service/2]).
-export([americano/0, cappuccino/0, tea/0, espresso/0,
  pay/1, cancel/0, cup_removed/0, open/0, close/0]).
```

注1 关于这一点我们真的不愿提及！


```
-export([stop/0, selection/3, payment/3, remove/3, service/3]).
-export([terminate/3, code_change/4]).
```

```
start_link() ->
```

```
    gen_fsm:start_link({local, ?MODULE}, ?MODULE, [], []).
```

```
...
```

```
cup_removed() -> gen_fsm:send_event(?MODULE,cup_removed).
```

```
open()          -> gen_fsm:send_event(?MODULE,open).
```

```
close()         -> gen_fsm:send_event(?MODULE,close).
```

```
...
```

在状态 *selection* 中，我们处理了 *open* 事件。这是唯一允许转换到我们的新 *service* 状态的状态 / 事件组合。在 *service* 状态下，收到 *close* 事件后，我们转换回 *selection* 状态。在其他所有状态下，*open* 和 *close* 事件都将被忽略。*service/3* 状态回调函数还处理同步 *stop* 事件，该事件停止 FSM 并触发对 *terminate/3* 的调用：

◀ 331

```
% State: drink selection
```

```
selection({selection, Type, Price}, LoopData) ->
```

```
    hw:display("Please pay:~w",[Price]),
```

```
    {next_state, payment, {Type, Price, 0}};
```

```
selection({pay, Coin}, LoopData) ->
```

```
    hw:return_change(Coin),
```

```
    {next_state, selection, LoopData};
```

```
selection(open, LoopData) ->
```

```
    hw:display("Open", [ 1]),
```

```
    {next_state, service, LoopData};
```

```
selection(_Other, LoopData) ->
```

```
    {next_state, selection, LoopData}.
```

```
% State: service
```

```
service(close, LoopData) ->
```

```
    hw:reboot(),
```

```
    hw:display("Make Your Selection", []),
```

```
    {next_state, selection, LoopData};
```

```
service({pay, Coin}, LoopData) ->
```

```
    hw:return_change(Coin),
```

```
    {next_state, service, LoopData};
```

```
service(_Other, LoopData) ->
```

```
    {next_state, service, LoopData}.
```

```
...
```

```
service(stop, _From, LoopData) ->
```

```
{stop, normal, ok, LoopData}.
```

...

我们现在需要实现新的 `code_change/4` 回调函数。在事件处理程序或通用服务器中调用时，此回调需要三个参数，而在 FSM 中调用时则需要四个参数：

```
Mod:code_change(Vsn, State, LoopData, Extra) ->
    {ok, NewState, NewLoopData} | %Finite State Machines
    {error, Reason}
Mod:code_change(Vsn, LoopData, Extra) ->
    {ok, NewLoopData} | %Generic Servers
    {error, Reason}
Mod:code_change(Vsn, LoopData, Extra) ->
    {ok, NewLoopData} | %Event Handler
    {error, Reason}
```

332 第一个参数 `Vsn` 在升级操作时对应的是源旧版本号，而在降级操作时则是目的旧版本号。在这个例子中它是 `1.0`，当它降级到以前的版本时它也可能是 `{down, 1.0}`。当模块没有版本指令时，可以使用 `md5` 模块校验和，当版本完全不重要时，可以使用通配符。

`State` 只会传给 FSM，其包含的是触发升级时 FSM 所处的状态。

最后两个参数包括循环数据和在此模块特定的升级指令中传递的任何额外参数。在我们的例子中，我们没有对 `_Extra` 参数做任何事情，也没有处理循环数据。

`code_change/4` 回调在成功时必须返回 `{ok, NewState, NewLoopData}`。而返回 `{error, Reason}` 将导致升级失败，并且——如果是 `gen_server` 或 `gen_fsm` 的话——触发节点重新启动以前的版本。而在事件处理程序中，返回 `{ok, NewLoopData}` 以外的任何内容或异常终止都将导致处理程序从事件管理器中删除，但节点将不会恢复到其以前的版本并重新启动。

这是我们的咖啡机 FSM 的 `code_change/4` OTP 回调函数的样子：

```
code_change('1.0', State, LoopData, _Extra) ->
    {ok, State, LoopData};
code_change({down, '1.0'}, service, LoopData, _Extra) ->
    hw:reboot(),
    hw:display("Make Your Selection", []),
    {ok, selection, LoopData};
code_change({down, '1.0'}, payment, {_Type, _Price, Paid}, _Extra) ->
    hw:return_change(Paid),
    hw:display("Make Your Selection", []),
```



```
{ok, selection, {}};
code_change({down, '1.0'}, State, LoopData, _Extra) ->
{ok, State, LoopData}.
```

与 Erlang 示例相比，我们仅稍微改变了该 behavior。无论处于何种状态，包括 *payment*，我们都不会更改循环数据并保持原来的状态。对于仅仅是添加功能或状态的场景，这很正常。而如果我们要将状态或循环数据随升级而更改，则也会位于此处。

如果升级故障触发降级并且处于 *service* 状态，则我们重新启动硬件并返回到 *selection* 状态，原因是版本 1.0 中不存在 *service* 状态。如果用户正在为咖啡付钱，我们会返回用户支付的全部金额并返回到 *selection* 状态。正如我们将看到的，降级会导致系统重启并从头启动旧版本。因此，如果你的旧版本依赖于某些在启动时设置并稍后更改的永久性值，请确保你的 `code_change` 将它们恢复为正确的值。

当完成新模块的实现时，我们将它们打包在一个 application 中，并升高版本号。在我们的例子中，新的 *coffee* application 的版本号是“1.1”，其中 *hw*、*coffee_app* 和 *coffee_sup* 等模块的版本也与 application 版本相同。同理，*coffee_fsm* 模块的版本现在也是 1.1。

◀ 333

如何升级 Record（记录）

BEAM 虚拟机中的记录（record）与我们在讨论数据库时常说的“记录”并非同一种数据结构。相反，在 BEAM 虚拟机内部，记录实际上是通过元组（tuple）来表示的，其中第一个元素对应记录的名称，其余元素则对应记录中原本的各个字段，其顺序符合记录中字段定义的顺序。如果在软件升级过程中涉及记录格式的变更，那么就必须使用记录的元组表示形式。但如果你使用的是映射组（map）而不是元组，则不会发生此问题。如果你不得不将其表示为元组的话，该如何应对？

回想一下我们的频率服务器，其中的记录的格式为：

```
-record(freq, {free, allocated})
```

当初始化后，其对应的元组表示看起来像这样：

```
{freq, [5,6,7,8], []}
```

假设在我们的升级中，我们想增加一个新字段用于屏蔽（block）某些频率，使得这些频率不允许被分配——即使它们并未被使用着。于是我们的新记录格式可能如下所示：

```
-record(freq, {free, allocated, blocked})
```

新版本模块中的 `code_change/3` 函数将处理不同记录版本的升级和降级问题，如下所示：

```
code_change('1.0', {freq, Free, Alloc}, _Extra) ->
    {ok, {freq, Free, Alloc, []}};
code_change({down, '1.0'}, {freq, Free, Alloc, Blocked}, _Extra) ->
    {ok, {freq, Free++Blocked, Alloc}}.
```

当需要更改 Mnesia 表格中的记录格式时，请使用 `mnesia:transform_table/3,4` 函数。它们会自动为表中执行转换的所有对象应用一个函数，使得你能够更改记录名称（不是表名称）并更新属性。

应用程序升级文件

我们的新版咖啡机 FSM 已经启动并运行了，现在需要一个 application 升级文件，其中包含在升级或降级到同一 application 的其他版本时要执行的一组操作。application 升级文件在概念上与 app 文件类似，因为 `systools` 会基于它们来创建升级脚本。它们的名称为 application 名加上 `.appup` 后缀，与 app 文件一起放置在 `ebin` 目录中。

进入安装 Erlang 的根目录并键入 `ls lib/*/ebin/*.appup`。该操作将返回随你安装 Erlang 发行包一同安装的所有 application 升级文件。自 Erlang/OTP 版本 17 开始，每个 application 都包含 `.appup` 文件。在此之前，你只能升级一些核心 application，因为并非所有 application 都提供了 `.appup` 文件。让我们来看看 2.6.1 版本的 `sasl.appup` 文件：

```
{"2.6.1",
%% Up from - max one major revision back
[{<<"2\\.[5-6](\\. [0-9]+)">>,[restart_new_emulator]}, % OTP-18.*
 {<<"2\\.[4](\\. [0-9]+)">>,[restart_new_emulator]}], % OTP-17
%% Down to - max one major revision back
[{<<"2\\.[5-6](\\. [0-9]+)">>,[restart_new_emulator]}, % OTP-18.*
 {<<"2\\.[4](\\. [0-9]+)">>,[restart_new_emulator]}] % OTP-17
}.
```

根据此内容，我们能弄清楚对于 2.6.1 版本的 application，当其在 OTP 版本 17 和 18 之间升级或降级时会发生什么。此 application 从版本 2.4.X、2.5.X 或 2.6 升级或降级到 2.6、2.5.X 或 2.4.X（其中 X 是补丁版本号），我们需要重新启动虚拟机（模拟器）。请注意，此处使用了（以二进制数据类型方式表示的）正则表达式，描述的不是单个版本号而是一个版本号范围，并提供了对应的升级和降级指令列表。除了正则表达式，你还可以使

用针对单个指定版本号的字符串，例如“2.4.5”。

检查你安装的发行包中的任何一个 *.appup* 文件，你将看到它们都遵循以下格式：

```
{Vsn,  
  [{UpFromV1, InstructionsU1}, ...,  
  {UpFromVK, InstructionsUK}],  
  [{DownToV1, InstructionsD1}, ...,  
  {DownToVK, InstructionsDK}]}
```

Vsn 是你要升级到的 application 版本。*UpFromV<N>* 是将被你升级的 application 版本。如果出现问题，*DownToV<N>* 是你可以降级到的 application 版本。*Vsn* 可以是具有确切版本号的字符串，也可以是包含正则表达式的二进制类型值，这使你能够针对多种 application 版本统一描述执行升级和降级时的指令。如果你安装了 OTP 版本 17 或更高版本，请查看各种 *.appup* 文件，你会注意到 OTP 标准 application 通常允许你升级或降级两个版本。

如果你计划使用正则表达式，则使用以下构造便足够表示版本范围了：

- 句点符号 (.) 匹配任意字符，因此表达式 1.3 将匹配从 1 启动并以 3 结尾的任何字符组合。
- 星号 (*) 表示匹配前一个元素零次或多次。
- 加号 (+) 表示匹配前一个元素一次或多次。
- 问号 (?) 表示匹配前一个元素零次或一次。
- 范围 [0-9] 表示匹配 0 至 9 的所有字符。
- 序列 \\ 表示一个普通句号。此处你需要转义反斜杠，是因为 Erlang 本身使用反斜杠来转义字符。
- 正则表达式中开头处的插入符号 (^) 起到锚定至开头的作用，使得针对版本字符串的开头做匹配。
- 正则表达式末尾的美元符号 (\$) 起到锚定至结尾的作用，使得针对版本字符串的末尾做匹配。

335

例如，`<<"^1\\. [0-9]+>>` 匹配任意的 1.X 版本，而 `<<"^1\\.0\\. [0-9]+>>` 匹配任意的 1.0.X 版本，`<<"^1\\. ([0-9]+\\.)?\\. [0-9]+>>` 则既能匹配 1.X 也能匹配 1.X.X，上述 X 都是整数。

如果不确定你写的正则表达式是否正确，请使用 `re:run(Vsn, RegExp)` 进行测试，如果匹配失败，则返回 `nomatch`，否则返回 `{match, MatchData}`。你可以在 `re` 模块的手册页中阅读有关正则表达式的更多信息。

浏览 `.appup` 文件，你应该会看到针对不同版本的行动列表。它们包括诸如 `restart_new_emulator`（仅在升级 `erts`、`kernel`、`stdlib` 和 `sasl` 应用时使用），`load_module`、`apply`、`restart_application` 和 `update` 等元素。在某些情况下，当不需要采取任何行动时，你会发现一个带有两个空列表的元组 `{Vsn, [], []}`。行动分为高级指令和低级指令。在创建发行版升级脚本时，高级指令将转换为低级指令。

回到我们的例子中，我们打算把咖啡机 FSM application 从 1.0 版升级到 1.1 版。这不会是一个复杂的升级，因为不涉及任何驱动程序或 NIF，没有新的 application 或模块添加到发行包中，并且不需要担心内部进程和模块间的依赖关系，更不用说内部状态或循环数据更改。在幕后，我们需要做的是暂停所有依赖模块 `coffee_fsm` 的 `behavior` 进程，加载模块的新版本，清除旧版本，调用 `code_change`，并恢复进程（参加图 12-3）。

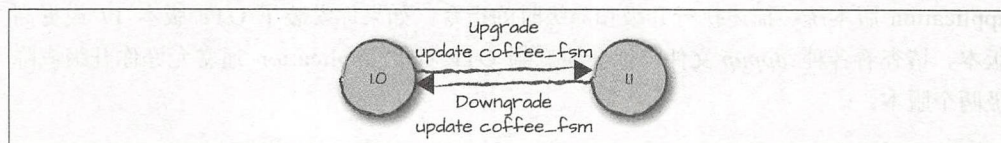


图 12-3: 咖啡机 FSM 版本转变

336 我们的 `coffee.appup` 文件中包含了一个元组，其中包含正在升级到的版本以及升级和降级相关的高级指令。在我们的例子中，`update` 用于加载新模块，`{advanced, {}}` 则触发 `code_change/4` 调用，其中传递了 `{}` 作为最后一个参数：

```
%% File:coffee.appup
{"1.1", % Current version
 [{"1.0", [{update, coffee_fsm, {advanced, {}}}], % Upgrade from
 [{"1.0", [{update, coffee_fsm, {advanced, {}}}], % Downgrade to
 }].
```

在升级和降级进程中，`update` 高级指令将转化为以下一系列低级指令：

1. 搜索目标模块的代码（object code），从文件中加载并缓存下来。这样做使得费时的文件操作能在暂停进程之前先完成。
2. 对于任何进程，只要它（在子进程规格中声明）依赖目标模块，便使用 `sys:suspend/1` 将其挂起。
3. 清除正在升级的模块的旧版本。
4. 加载模块的新版本，并使 `current` 版本成为 `old` 版本。
5. 清除该模块的 `old` 版本，该版本在步骤 4 之前原本是 `current` 版本。

6. 调用 `Mod:code_change/4`。

7. 使用 `sys:resume/1` 恢复先前挂起的进程，允许它们继续处理新的请求。

到目前为止，一切顺利，但我们是如何将模块依赖关系映射到具体的 `behavior` 进程的呢？还记得吗，在监督者子进程规格中，你必须列出实现该 `behavior` 所涉及的模块：

```
{coffee_fsm, {coffee_fsm, start_link, []},  
  permanent, 5000, worker, [coffee_fsm]}
```

之所以必须列出它们，正是在因为在升级或降级期间，`systools` 要告诉监督者在升级一个或多个核心模块时暂停某些指定的进程。而对于事件处理器以及其他特殊进程——它们的模块清单在编译时尚不可知——我们将用原子 `dynamic` 替换模块列表，然后当需要做升级时，再查询进程来获知。

◀ 337

出于可伸缩性原因，OTP 需要区分动态和静态模块集。倘若我们每次进行软件升级时，为了筛选出哪些进程不包含正在升级的模块，而必须询问数百万个 `behavior` 进程是毫无意义的。通常使用动态模块的进程很少，因而在进行升级时很少影响性能。但如果你用到了动态子进程，并且你清楚将有数百万个实例同时并存而且在编译时无法确定模块列表，那么你得自己选择一种可伸缩的升级策略，或者决定根本不做任何升级。

高级指令

`.appup` 文件中的操作分为高级 (high-level) 指令和低级 (low-level) 指令，当生成升级脚本时，高级指令实际上会转换为低级指令。为了简单（和理智）起见，我们鼓励你尽可能使用高级指令并避免低级指令，即使它们可以混合在一起。让我们更详细地看一下高级指令的细节。

`{update, Mod}`

该指令及其所有变体用于执行同步式代码替换 (synchronized code replacements)，其中依赖于 `Mod` 的所有进程必须在加载新版本模块之前暂停。当新版本模块被加载并清除旧版本后，会恢复挂起的进程。这是模块更新命令的最简单变体，因为其不会调用 `code_change/3,4` `behavior` 回调。如果希望所有进程始终与其他相交交互的进程表现一致，则你需要同步并挂起依赖于 `Mod` 的所有进程。如果你在加载新模块之前没有暂停它们，则某些进程可能会表现为旧的行为，而其他某些进程则会表现为新的行为。

`{update, Mod, supervisor}`

如果 `Mod` 是监督者回调模块，并且你想更改 `init/1` 回调函数返回的监督者规格描

述,则需要使用此高级指令。如果要添加子进程,则需要使用 `supervisor:start_child/2` 函数处理监督树中的任何更改。如果你要移除子进程,请使用 `supervisor:terminate_child/2` 和 `supervisor:delete_child/2`。我们在第 8 章的“动态子进程”一节中介绍了这些功能。如果你要更改由 `rest_for_one` 依赖关系而启动的子进程的顺序,则更新会变得更加复杂。你必须按照 `init/1` 回调函数中指定的顺序终止子进程并重新启动它们。

338 > `{update, Mod, {advanced,Extra}}, {update, Mod, DepMods}, {update, Mod, {advanced,Extra}, DepMods}`

如果我们包含 `{advanced,Extra}` 元组,则升级脚本将调用 `Mod:code_change/3,4` 回调函数,并将 `Extra` 作为最后一个参数传递。当升级需要更改 `behavior` 状态和循环数据时,你将需要此选项。对于这个以及所有其他 `update` 指令,可以省略 `{advanced,Extra}` 或将其替换为 `soft`,这两种做法都会导致 `code_change` 不会被调用。`DepMods` 对应的是 `Mod` 所依赖的模块列表。使用这些模块的 `behavior` 也将被暂停。

`{update, Mod, {advanced,Extra}, PrePurge, PostPurge, DepMods}`

`PrePurge` 和 `PostPurge` 默认被设置为 `brutal_purge`。如果希望运行旧版本 `Mod` 的进程在新版本加载之前无条件终止,以及在发行包永久化之后进行模块升级,都可以使用此选项。你可以通过将 `PrePurge` 设置为 `soft_purge` 来覆盖此行为。如果某些进程仍在运行老版本的代码,则触发执行 `relup` 文件的 `release_handler:install_release/1` 将会返回 `{error,{old_processes,Mod}}`。如果 `PostPurge` 被设置为 `soft_purge`,则只有在执行老版本代码的进程终止其调用后,发行包处理程序才会清除 `Mod`。

`{update, Mod, Timeout, {advanced,Extra}, PrePurge, PostPurge, DepMods}`

还记得吗, `behavior` 是基于回调函数实现的,因此要使清除模块出错,它们必须在回调中执行非常长的时间,或者拥有非常长的消息队列。尝试挂起进程时的默认超时值为 5 秒,但可以通过将 `Timeout` 字段设置为以毫秒为单位的整数或原子 `infinity` 来覆盖此值。如果 `behavior` 不响应 `sys:suspend/1` 调用,并且超时被触发,则该进程将被忽略。如果进程正在执行的模块被清除,它可能会被终止,或者也可能当进程启动后运行新版模块却未能顺利执行升级过程而导致出现运行时错误时也会被终止。当你在重负载下测试升级过程后,你可能会发现需要增加超时值,因而你可能会使用到 `Timeout` 选项。

`{update, Mod, ModType, Timeout, {advanced,Extra}, PrePurge, PostPurge, DepMods}`

默认情况下, `code_change/3,4` 回调函数之一会在新模块加载后执行。而在降级的情况下, 则是在加载模块之前调用 `code_change/3,4`。你可以通过将 `ModType` 设置为 `static` 来覆盖此设置, 这会在升级或降级之前加载该模块并调用 `code_change/3,4`。如果未指定, 或者需要默认 behavior, 请将 `ModType` 设置为 `dynamic`。

`{load_module, Mod}, {load_module, Mod, DepMods}, {load_module, Mod, PrePurge, PostPurge, DepMods}` 339

你可能想在不需暂停进程的情况下使用此低级指令进行升级。我们将这种升级形式称为简单代码替换 (simple code replacements)。这同样适用于添加和删除模块的指令。`DepMods` 列出了加载 `Mod` 之前应该加载的所有模块。这个参数默认是一个空列表。`PrePurge` 和 `PostPurge` 可以设置为 `soft_purge` 或 `brutal_purge` (默认值)。它们的工作方式与使用 `update` 命令的方式相同, 可在处理库模块或扩展不影响正在运行的进程的功能时使用此指令。

`{add_module, Mod}, {delete_module, Mod}`

这些命令将转换为在不同版本发行包之间添加和删除模块的低级指令。

`{add_application, Application}, {add_application, Application, Type}`

该指令将为新版本发行包添加一个新的 `application`, 并加载 `app` 文件中定义的所有模块, 并在适用的情况下启动监督树。第 9 章曾介绍过 `application` 的类型, 此处默认为 `permanent`, 但 `Type` 也可以设置为 `transient`、`temporary`、`load` 或 `none`。

`{remove_application, Application}, {restart_application, Application}`

在删除或重新启动 `application` 时, 你将希望使用这些命令。移除 `application` 会关闭监督树, 从内存中删除模块并停止 `application`。如果升级或降级需要重新启动 `application`, 则此高级命令将转换为停止并启动 `application` 及其监督树的低级命令。通常你在 `.appup` 文件中碰到的重启 `application` 的情形中, 针对的主要是非核心型 OTP `application`, 它们通常是工具和库 `application`, 重新启动它们可以不影响在线系统流量。

你可以在同一个 `.appup` 文件中混合使用高级指令和低级指令, 但对绝大多数用例而言, 高级指令就足够了, 因为你的大部分操作都可以使用它们完成。在下一节我们完成第一次升级操作后, 会继续介绍低级指令。

发行包升级文件

现在我们有 *coffee.appup* 文件，并理解了高级指令的功能，让我们使用这些知识来生成升级包。第一步是使用 `systools:make_script/2` 创建一个新的启动文件。它本身并不用于升级，但它是我们部署的软件包的一部分，预防升级后节点必须重新启动（无论出于何种原因）。在第二个 shell 命令中，我们创建一个名为 *relup* 的发行包升级文件，该文件位于当前工作目录中。该文件使用 *rel* 和 *.appup* 文件中指定的模拟器和 application 版本生成，使用它们可获取 *.appup* 文件中的高级和低级指令并将其映射为一系列低级指令。编译你的 *coffee-1.1* application 目录中的所有代码，并运行以下命令：

```
1> systools:make_script("coffee-1.1", [{path, ["coffee-1.1/ebin"]}]).
```

ok

```
2> systools:make_relup("coffee-1.1", ["coffee-1.0"], ["coffee-1.0"],  
    [{path, ["coffee*/ebin"]}]).
```

ok

```
3> systools:make_tar("coffee-1.1",  
    [{path, ["coffee-1.1/ebin"]},  
    {outdir, "ernie/releases"}]).
```

ok

在第三个 shell 命令中，我们创建了 tar 文件 *coffee-1.1.tar.gz*。它包含了 *coffee-1.1.rel* 中所指定的 *lib* 和 *releases* 目录。调用 `make_tar/2` 会自动拾取 *relup*、*start.boot* 和 *sys.config* 文件，并在 *releases* 下创建发行包 1.1 的目录。请注意，与我们的第一次安装不同，我们未包含 *erts* 选项。因为我们打算使用已安装的 Erlang 模拟器。

现在让我们仔细查看 *relup* 文件，其中已经生成了低级指令。我们将在本章后面的“低级指令”一节解释它们，但即使不介绍它们，你也应该很容易弄清楚：

```
{"1.1",  
  [{"1.0", [],  
    [{load_object_code, {coffee, "1.1", [coffee_fsm]}},  
     point_of_no_return,  
     {suspend, [coffee_fsm]},  
     {load, {coffee_fsm, brutal_purge, brutal_purge}},  
     {code_change, up, [{coffee_fsm, {}}]},  
     {resume, [coffee_fsm]}]}],  
  [{"1.0", [],  
    [{load_object_code, {coffee, "1.0", [coffee_fsm]}},  
     point_of_no_return,  
     {suspend, [coffee_fsm]},  
     {code_change, down, [{coffee_fsm, {}}]}],
```



```
{load,{coffee_fsm,brutal_purge,brutal_purge}},  
{resume,[coffee_fsm]}}}}).
```

在详细介绍低级命令之前，让我们看看用来生成文件本身的 `systools:make_relup/3,4` 调用：

```
systools:make_relup(RelName, UpFromList, DownToList, [Options]) ->  
    ok | error | {ok,Relup,Module,Warnings} | {error,Module,Error}
```

该调用需要 `RelName`，我们正在升级或降级的发行包的名称。它指向 `RelName.rel` 文件，用于确定 Erlang 运行时系统和各种 application 的版本。`RelName` 也可以是元组 `{RelName, Descr}`，其中 `Descr` 是一个 Erlang 数据项，在升级和降级指令中用到，由在目标机器上安装发行包的函数返回。

341

第二个参数和第三个参数 `UpFromList` 和 `DownToList` 分别包含我们要升级或降级到的发行包的列表。它们都是一些名称，对应特定版本的 `rel` 文件，用于确定需要添加、删除或升级哪些 application。使用它们各自的 `.app` 和 `.appup` 文件，调用还能决定需要执行的命令的顺序。第四个可选的参数是一个选项列表，可能的选项如下所述。

`{path, DirList}`

将 `DirList` 中列出的路径添加到代码搜索路径。你可以在路径中包含通配符，因此 `"lib/*/ebin"` 中的星号将展开包含 `lib` 中所有包含 `ebin` 子目录的目录。创建 `relup` 文件的节点的代码搜索路径中必须包含指向 `.rel` 和 `.app` 文件的旧版本和新版本的路径，并包含指向新的 `.appup` 和 `.beam` 文件的路径。

`{outdir, Dir}`

将 `relup` 文件放到 `Dir` 目录中而不是当前工作目录。

`restart_emulator`

生成升级或降级后重新启动节点的低级指令。

`silent`

返回格式为 `{ok, Relup, Module, Warnings}` 或 `{error, Module, Error}` 的元组，而不是将结果打印到 I/O。从脚本或集成构建过程中调用 `systools` 函数并在需要处理错误时使用此选项。

`noexec`

返回与 `silent` 选项相同的值，但不生成 `relup` 文件。

warnings_as_errors

将警告视为错误，并在发生警告时拒绝生成 *relup* 脚本文件。

relup 文件的格式与 *.appup* 文件类似：

```
{Vsn,  
  [{UpFromV1, Descr, InstructionsU1}, ...,  
  {UpFromVK, Descr, InstructionsUK}],  
  [{DownToV1, Descr, InstructionsD1}, ...,  
  {DownToVK, Descr, InstructionsDK}]}.
```

Descr Erlang 数据项来自 `systools:make_relup/3,4` 调用时传递的 `{RelName, Descr}` 元组。如果 Descr 在调用中被省略，则默认为空列表。你将在我们的示例中看到这一点，因为我们将它留给了咖啡机 *relup* 示例。当在目标机器上自动安装升级时，Descr 变得有意义，因为它的值可以被安装升级的程序或脚本使用。

低级指令

Relup 文件由从 *.appup* 文件生成的低级 (low-level) 指令集组成。对于复杂的升级，你可以使用低级指令编写文件或手动编辑生成的文件。低级指令包含以下内容。

`{load_object_code, {Application, Vsn, ModuleList}}`

读取 `Application ebin` 目录中的所有模块，但不会将它们加载到运行时系统中。该指令在暂停 `behavior` 和特殊进程之前执行。这不同于高级指令 `load`，它不仅加载模块，而且使其在运行时系统中可用。

`point_of_no_return`

该指令在 *relup* 脚本中只应出现一次，并应放置在系统无法执行 *relup* 文件中的一条或多条指令、出错无法恢复处。执行该指令后发生的崩溃将导致重新启动系统的旧版本。它通常放在 `load_object_code` 指令之后。

`{load, {Module, PrePurge, PostPurge}}`

使某个已被 `load_object_code` 加载的模块成为 `current` 版本。`PrePurge` 和 `PostPurge` 可以设置为 `soft_purge` 或 `brutal_purge` (默认值)。

`{apply, {Mod, Func, ArgList}}`

调用 `apply(Mod, Func, ArgList)`。如果 `apply` 在“不返回点” (the point of no return, 参见前面的 `point_of_no_return` 指令) 之前执行并且出故障或返回 (或抛出) `{error, Error}`，则对 `release_handler:install_release/1` 的

调用将分别返回 `{error,{'EXIT',Reason}}` 或 `{error,Error}`。如果它在不返回点之后执行并且出故障，则系统将使用旧版本的发行包重新启动。该指令可以用来代替 `code_change/3,4` 回调函数。

`{remove, {Module, PrePurge, PostPurge}}`

与 `load` 和 `purge` 一起使用。该指令使 `current` 版本的 `Module` 变为 `old` 版本。

`{purge, ModuleList}`

清除 `ModuleList` 中所有模块的旧版本。执行旧版代码的 `behavior` 和特殊进程被终止。

`{suspend, [Module | {Module, Timeout}]}`

挂起依赖于 `Module` 列表的 `behavior` 进程。`Timeout` 是以毫秒为单位的整数或原子 `default` (设置为 5 秒) 或 `infinity`。如果对 `sys:suspend/1` 的调用未在 `Timeout` 内返回，则该进程将被忽略但不会终止。

`{resume, ModuleList}`

恢复依赖于 `ModuleList` 中列出的模块的已暂停进程。

`{code_change, [{Module, Extra}], {code_change, Mode, [{Module, Extra}]}`

触发 `Module:code_change/3,4` 调用，并在运行 `Module` 的所有 `behavior` 进程中传递 `Extra`。`Mode` 是 `up` 或 `down`，决定了是升级调用还是降级调用。如果省略，则 `Mode` 默认为 `up`。

`{stop, ModuleList}`

该指令导致针对所有依赖于 `ModuleList` 中指定的模块之一的 `behavior` 进程调用 `supervisor:terminate_child/2`。

`{start, ModuleList}`

通过调用 `supervisor:restart_child/2` 启动所有依赖于 `ModuleList` 中某个模块的已停止的进程。

`restart_new_emulator`

升级模拟器或 `kernel`，`stdlib` 和 `sasl` 核心 `application` 时使用此指令。在升级这些 `application` 之后，但在执行 `relup` 文件的其余部分之前，需要立即重新启动模拟器。所有其他 `application` 将在新模拟器中运行旧版本时重新启动，并在新模拟器中运行 `relup` 文件的其余部分时升级。当不同的进程以这种方式运行不同的 `application` 版本时，它们之间可能会发生非后向兼容性冲突，因此在使用此技术之前，请确保升级进程涉及的所有可能场景都已经过适当测试。如

果你担心低级指令的顺序, 请使用高级指令并让 `systools:make_relp/3,4` 生成 `relup` 文件。升级进程中只能执行一次该指令。

`restart_emulator`

在不涉及核心 `application` 或模拟器升级的升级中, 需要模拟器重启时使用此指令。它在 `relup` 文件中只能出现一次, 并且必须是最后一条指令。

安装升级

让我们将目光放回到我们生成的 `coffee-1.1.tar.gz` 文件上, 接下来将用它完成在线升级。假设它已被放置在目标环境的 `releases` 目录中。从 `ernie` 根目录中, 连接到运行着 1.0 版本 `coffee_fsm` 的节点。如果未运行, 请用 `bin/start` 启动。我们调用 `release_handler:unpack_release/1` 解压新版本, 从而解压缩得到所有文件, 将 `coffee-1.1 application` 添加到 `lib` 目录, 并在 `releases` 目录中创建 1.1 版的目录。我们可以在 shell 命令 2 和 3 中看到, 在解压缩新版本后, 它与 1.0 并列, 并且 1.0 仍在运行:

```
$ bin/to_erl /tmp/
```

```
Attaching to /tmp/erlang.pipe.1 (^D to exit)
```

```
1> release_handler:unpack_release("coffee-1.1").
```

```
{ok, "1.1"}
```

```
2> release_handler:which_releases().
```

```
[{"coffee", "1.1",  
  [{"kernel-4.1.1", "stdlib-2.7", "sas1-2.6.1", "coffee-1.1"},  
   unpacked},  
 {"coffee", "1.0",  
  [{"kernel-4.1.1", "stdlib-2.7", "sas1-2.6.1", "coffee-1.0"},  
   permanent]}]
```

```
3> application:which_applications().
```

```
[{coffee, "Coffee Machine Controller", "1.0"},  
 {sas1, "SAS1 CXC 138 11", "2.6.1"},  
 {stdlib, "ERTS CXC 138 10", "2.7"},  
 {kernel, "ERTS CXC 138 10", "4.1.1"}]
```

```
4> coffee_fsm:espresso().
```

```
Display: Please pay:150
```

```
ok
```

```
5> coffee_fsm:pay(100).
```

```
Display: Please pay:50
```

```
ok
```

```
6> release_handler:install_release("1.1").
```

```
{ok, "1.0", []}
```

```
7> coffee_fsm:cancel().
```

```
Display: Make Your Selection
```



```

ok
Machine:Returned 100 in change
8> coffee_fsm:open().
ok
Display:Open
9> coffee_fsm:close().
Machine:Rebooted Hardware
Display:Make Your Selection
ok
10> application:which_applications().
[{coffee,"Coffee Machine Controller","1.1"},
 {sas1,"SASL CXC 138 11","2.6.1"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]
11> init:restart().
ok
12>
Erlang/OTP 18 [erts-7.2] [smp:8:8] [async-threads:10] [kernel-poll:false]

```

345

...<snip>...

```

Eshell V7.2 (abort with ^G)
1> application:which_applications().
[{coffee,"Coffee Machine Controller","1.0"},
 {sas1,"SASL CXC 138 11","2.6.1"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]

```

接下来，我们通过执行 `release_handler:install_release/1` 调用来升级发行包。如果出现问题并且重新启动被触发，系统将重新启动并恢复到旧发行包。如果系统稳定，则通过调用 `release_handler:make_permanent/1` 将当前新发行包永久化。

然后，使用我们添加的新客户端函数来测试状态 *service* 的转入与转出，然后再重新引导 shell 命令 11 中的节点。因为我们未将发行包永久化，所以节点重新启动后发行包将恢复 1.0 版本。

接下来，在 shell 命令 2 和 3 中，我们重新安装该发行包并将其永久化。此刻，我们不再需要 1.0 版相关的文件了。可以调用 `release_handler:remove_release/1` 将不再使用的发行包从系统中删除。该调用只会删除 *lib* 目录中那些仅属于指定版本发行包的 *application*，并从 *releases* 中删除对应目录，并在那里更新 *RELEASES* 文件。要恢复到旧版发行包，我们只能重新安装，涉及我们先前描述的所有步骤，包括为 *coffee* application 的 1.0 版创建 *.appup* 文件、*relup* 文件和 *tar* 文件：

```

2> release_handler:install_release("1.1").
{ok,"1.0",[]}
3> release_handler:make_permanent("1.1").
ok
4> release_handler:remove_release("1.0").
ok
5> release_handler:which_releases().
[{"coffee","1.1",
  ["kernel-4.1.1","stdlib-2.7","sasl-2.6.1","coffee-1.1"],
  permanent}]
6> halt().
[End]
$ ls lib/
coffee-1.1 kernel-4.1.1 sasl-2.6.1      stdlib-2.7

```

就是这样！你的软件已经实现了运行时升级功能，并具备发生问题时回退到旧发行包，或在不再需要时将它们删除的能力。

346



发行包处理程序主要是针对嵌入式目标系统（embedded target system）设计的。如果将其用于简单目标系统（simple target system），则需要确保重启模拟器时使用正确的引导和配置文件。具体如何做有很大的自由度；你可以替换现有的文件，也可以设置 OS 环境变量指向正确的文件。

发行包处理器

我们在第 9 章介绍了 SASL 应用。它是每个发行版必须包含的核心 OTP application 之一，因为它提供了包括构建、安装和升级发行包所需的系列工具。如果查看 SASL 的监督树（参见图 12-4），你可能已经注意到 release_handler（发行包处理程序）进程。它负责在每个节点上解压缩、安装和升级发行包。它还能将发行包移除或使其永久化。我们已经使用过发行包处理程序并在示例中实践过这些阶段。

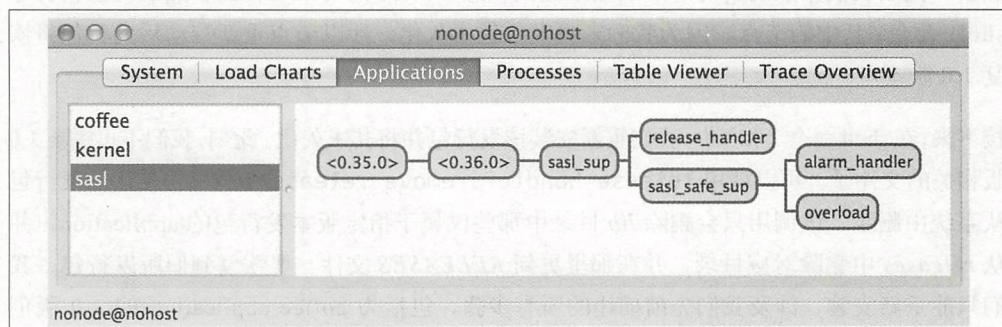


图 12-4：发行包处理器（release handler）进程

发行包处理程序假定你使用 `systools:make_tar/1,2` 创建的 tar 文件放置在 `releases` 目录中。每个发行版的发行包可以处于以下状态之一，如图 12-5 所示：`unpacked`、`current`、`permanent` 和 `old`。状态转换发生在 `release_handler` 模块中的函数被调用时，或者未永久化的发行包出错触发系统重新启动时。在任何时候，总是有一个发行包要么处于 `current` 状态，要么处于 `permanent` 状态。让我们来看看 `release_handler` 模块导出的函数，包括那些与触发转换紧密相关的函数。

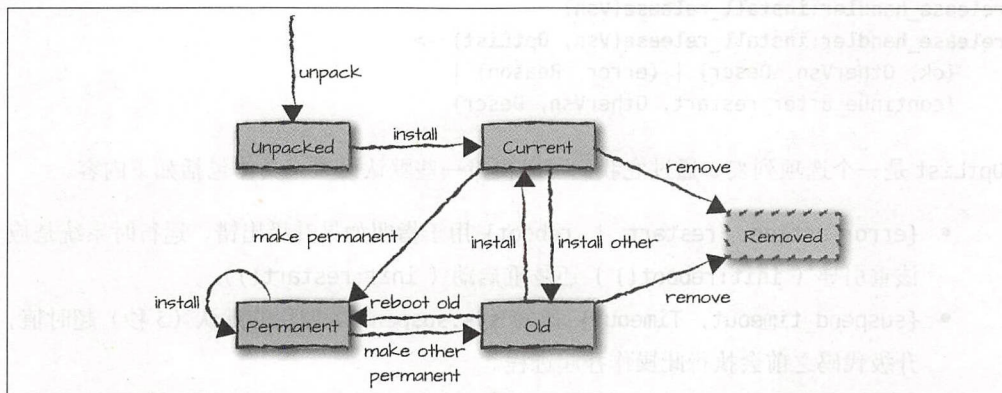


图12-5: 管理发行包

在处理你的第一份目标安装（target installation）时，只有在目标机器上已经安装了 Erlang 时，才能使用发行包处理程序。因为当我们创建第一个 `coffee_fsm` 发行包时，并不满足这一点，因此所有事情都必须手动完成。如果你按照这些步骤操作，会注意到，在系统的 1.0 版启动运行后我们执行的第一个调用是创建 `RELEASES` 文件：

```
release_handler:create_RELEASES(Root, RelDir, RelFile, AppDirs) ->
    ok | {error, Reason}
```

此调用将创建第一个版本的 `RELEASES` 文件，存储在 `releases` 目录中。该文件包含发行包处理程序的持久化状态，包括发行包 `application`、它们的版本以及它们的绝对路径。执行此功能的 Erlang 虚拟机必须具有写入 `releases` 目录的权限。`Root` 是 Erlang 的根目录，而 `RelDir` 是指向 `releases` 目录的路径。`releases` 目录通常位于 Erlang 根目录中，但可以通过设置第 11 章“发行包目录结构”一节中描述的 OS 或 OTP 环境变量来覆盖此目录设定。`RelFile` 指向位于 `releases` 目录中的 `rel` 文件，而 `AppDirs` 是格式如 `{App, Vsn, Dir}` 的元组列表，用于覆盖存储在 `lib` 中的 `application` 的值。当需要把 Erlang 打包分发至特定操作系统时常常使用此函数。下面的函数能够解压位于 `releases` 目录中的 `Name.tar.gz` 文件：

```
release_handler:unpack_release(Name) ->
```

```
{ok, Vsn} | {error, Reason}
```

它会检查是否所有必需的文件和目录都存在，并已将这些 application 添加到 *releases* 下的 *lib* 和 *release* 目录中。如果字符串 *Name* 是现有的发行包，或者在解压缩或读取必需的文件和目录时出现问题，则会失败。当我们解压缩发行包时，*install_release/1,2* 会触发软件升级（或降级），执行 *relup* 文件中指定的指令：

```
348 > release_handler:install_release(Vsn)
release_handler:install_release(Vsn, OptList) ->
    {ok, OtherVsn, Descr} | {error, Reason} |
    {continue_after_restart, OtherVsn, Descr}
```

OptList 是一个选项列表，通过它我们可以覆盖一些默认设置。具体包括如下内容。

- *{error_action, restart | reboot}* 用于指明如果升级出错，运行时系统是应该重引导（*init:reboot()*）还是重新启动（*init:restart()*）。
- *{suspend_timeout, Timeout}* 改写 *sys:suspend/1* 调用的默认（5 秒）超时值，升级代码之前会执行此操作挂起进程。
- *{code_change_timeout, Timeout}* 覆盖 *sys:change_code/4* 调用的默认（5 秒）超时，用于通知暂停的进程升级代码。
- *{update_paths, Bool}* 覆盖 *create_RELEASES/4* 调用中 *AppDirs* 参数中提供的默认目录 *lib/App -Vsn* 时使用。将 *Bool* 设置为 *true* 将导致更改 *AppDirs* 中 *application* 的所有代码路径，包括未升级的 *application*。将其设置为默认值 *false* 将只会导致升级后的 *application* 的路径发生更改。

你可能会记得 *relup* 文件中包含格式为 *{Vsn, Descr, Instructions}* 的元组。其中 *Descr* 是升级或降级成功时返回值的一部分。如果返回 *{continue_after_restart, OtherVsn, Descr}*，则表明正在升级运行时系统和核心 *application*，需要先重新启动模拟器才能执行剩余的脚本。

如果发生了可以从中恢复的错误，则 *{error, Reason}* 被返回。可恢复错误包括 *Vsn* 已永久化或缺少 *relup* 文件，以及其他将导致发行包安装故障但不需要节点重新启动的其他错误。如果升级由于不可恢复的错误而出现故障，则该节点将重新启动或重新引导。

安装发行包和升级代码可能是一项有风险并且耗时的操作。此功能可缓解发生问题的风险，检查是否可安装 *Vsn*，确保所有必需的文件均可用并可访问，并执行 *relup* 文件中 *point_of_no_return* 之前的所有低级指令：

```
release_handler:check_install_release(Vsn)
```



```
release_handler:check_install_release(Vsn,Options) ->
    ok | {error, Reason}
```

Options 是一个包含 [purge] 的列表，表示在检查过程中软清除 (soft purge) 代码。这将加快发行包本身的安装，因为在升级之前所有模块都会被软清除。 349

当安装好新发行包并执行了 *relup* 文件中的指令后，我们将观察节点，并可能运行一些诊断性测试。如果发现问题，便会使用旧的引导文件重新启动节点，使得重新启用旧发行包。调用 *make_permanent/1* 会改变重新引导或重新启动节点时使用的引导脚本为升级时所用的新脚本。此调用可能因为各种原因失败，包括如 Vsn 不是当前版本，或者当前根本不是发行包等：

```
release_handler:make_permanent(Vsn) -> ok | {error, Reason}
```

如果发行包已永久生效，则可以删除指定的旧发行包文件了。调用 *remove_release/1* 将删除不再使用的旧 application，并删除 *releases/Vsn* 目录下包含 *.rel*、*.boot* 和 *sys.config* 文件的指定 Vsn 目录。此调用还会更新 *RELEASES* 文件中的可用发行包信息。如果 Vsn 是已永久化的或不存在的发行包，则会出错：

```
release_handler:remove_release(Vsn) -> ok | {error, Reason}
```

如果你的当前发行包未按预期运行，并且需要恢复到旧发行包（假设你尚未删除），则此调用会使用旧的启动文件重新启动运行时系统，并永久化使其成为当前采用的版本：

```
release_handler:reboot_old_release(Vsn) -> ok | {error, Reason}
```

此调用使用 *RELEASES* 文件并返回发行包处理器已知的所有发行包列表。Status 是 unpacked、current、permanent 或 old 之一：

```
release_handler:which_releases(Status)
release_handler:which_releases() -> [{Name, Vsn, Apps, Status}]
```

release_handler 模块导出的函数中有一些可用于升级和降级单个 application，并可快速创建发行升级脚本并执行。这些函数（我们在本书中未涉及）旨在促进和自动化针对 application 升级过程的测试。它们不应该用于生产系统，因为这些更改不会持久，系统重启便失效了。

在不使用发行包处理器的情况下安装升级，并维持一致性且及时更新是可行的。但是发行包处理器提供的功能对于处理操作系统特定的软件包，使用其他工具进行部署和升级，或者甚至编写自己的软件包，都很有用。借助其中某些函数，我们可以通知发行包处理器进程添加和删除发行包和与发行包相关的文件。在随标准 Erlang 发行包附带

的 `release_handler` 手册页中可以阅读到这些函数的具体介绍，并了解其升级和降级单个 application 的能力。

350

升级环境变量

当升级发行包时，新的包中将包含新的（同时也是必需的）`sys.config` 文件。并且针对每一个新的或者更新的 application 还会包含一个新 app 文件。这些文件可能包含新的或更新的 application 环境变量，而如果 app 文件不再需要，它们将被完全省略。在升级过程中，application 控制器会把旧的环境变量与启动脚本中的当前（使用 `- application key value` 标志设置的）环境变量进行比较，并且还会比较配置文件、app 文件，然后相应地针对差异做更新。完成后，在恢复进程之前，会调用新的 application 的回调模块中的如下回调函数：

```
Module:config_change(Updated, New, Deleted)
```

Updated、New 和 Deleted 都是由格式为 {Key, Value} 的元组构成的列表，其中每个 Key 都对应一个环境变量，并且 Value 正是你希望设置为的值。这是一个可选的回调，可以省略，但如果进程状态依赖于启动时读取的环境变量，那么这一功能就变得非常有用。

执行发行包永久化操作会将启动脚本指向的 `sys.config` 文件更改为指向新版本。做完这一步才算完成，因为如果未执行永久化，则重新启动节点时将会自动恢复到以前的版本。

升级特殊进程

升级特殊进程与升级普通 behavior 没有区别。如果简单代码替换即可满足，可使用 `add_module` 指令来加载新模块。如果必须采用同步代码替换，请使用 `update` 这一高级指令，我们升级 OTP behavior 时用到的也是此指令。一旦收到格式为 {system, From, Msg} 的消息后，特殊进程会调用 `proc_lib:handle_system_msg/6`，导致该进程被暂停。（我们在第 10 章的“系统消息”一节中介绍过相关内容。）如果 `update` 命令的 `Change` 字段中包含 {advanced, Extra} 参数，则会调用特殊进程回调模块中的以下回调函数：

```
Mod:system_code_change(LoopData, Module, Vsn, Extra) ->
    {ok, NewLoopData}
```

此调用返回元组 {ok, NewLoopData}。Module 是回调模块的名称，Vsn 是要升级到的版本，或者在降级情况下是 {downgrade, Vsn}。Vsn 在这两种情况下都是字符串。

最后一个注意事项是，还记得系统消息 `{get_modules, From}` 吗？当特殊进程不知道它所依赖的模块时，它必须处理这条系统消息。在第 10 章的“动态模块和休眠”一节中我们曾介绍过这一点，在这种情况下我们会在监督者子进程规格中使用 `dynamic` 原子，你是否还有印象？在升级时，监督者中所有此类在子进程规范中将模块依赖关系设置为 `dynamic` 的进程都必须负责回复 `From!{modules, ModuleList}` 这样的消息，其中包含特殊进程当前所依赖的模块列表。这样一来，发行包处理程序才能知道应如何协调同步升级，以确定此特殊进程是否属于依赖链的一部分，进而确定是否应在某个模块升级期间暂停该特殊进程。

在分布式环境下升级

在分布式环境中做同步式软件升级？那可能吗？我们疯了才会去这样尝试？然而如果你拥有一个小型集群，并且你的网络可信，而且升级时存在依赖关系的节点间都已相互连接，那么为什么不能呢？请记住，分布式 Erlang 最初是为在同一个数据中心内运行并且位于防火墙后的集群而设计的，而且通常位于同一个机架中。如果你正在升级交换机，而事实上分布式 Erlang 节点通常运行在同一个交换机的控制背板上，所以如果你的网络出了故障，那么你也就失去了一切控制力，事实上完全失去了你的交换机。

对于运行了少量节点并且这些节点都运行在同一机架中的小型集群，你几乎不用担心。但对于较大的集群，例如跨数据中心的集群或网络不可靠的集群，则设计节点升级策略时是不会设计为同步式方案的。

警告已经够多了。让我们喝一些红牛 (Red Bull)，继续。如果在 `.appup` 文件中包含 `sync_nodes` 低级指令，则生成的 `relup` 脚本将与等待其他也希望同步升级的节点，并在同步达成时进行升级。

同步 (Synchronization) 是由以下指令之一触发的：

```
{sync_nodes, Id, NodeList}
{sync_nodes, Id, {Mod,Func,ArgList}}
```

你可以像第一条指令那样在 `.appup` 文件中硬编码 `NodeList`，或者使用第二条指令调用 `apply(Mod, Func, ArgList)` 以获取能识别 `Id` 的节点的列表，这些节点是要同步的节点。`Id` 可以是任何有效的 Erlang 数据项。要使同步成功，远程节点必须使用相同的 `Id` 执行相同的指令。

如果由于网络分区或因远程节点崩溃而导致你与尝试进行同步的远程节点失去连接，则

将使用旧发行包重新启动节点。此过程无超时机制，所以如果远程节点没有升级或不同步，尝试升级的本地节点将挂起，直到所有远程节点执行完 `sync_nodes` 或者其中某个节点的连接丢失。这就是为什么本节中介绍的技术对于远距离广域网络环境下的节点有一定的风险的原因。

352 如果尚未同步成功，则集群将挂起并等待其他节点。如果你的网络连接出现问题，或其他节点中的某个节点的升级出现故障，则会触发一系列节点重启，希望恢复并继续运行旧发行包。然而在最糟糕的情况下，这种技术可能会导致级联故障，因为当它们无法应对重新启动时，会将节点一个接一个地敲出。记住我们在此给你的警告！只有在安全且使用场景相符的情况下才能使用同步分布式升级。如果无法确定是否能满足此条件，则应在确保运行新发行包的节点能够与仍在运行旧发行包的节点进行互操作之后，再在集群中一次一个节点地执行滚动式升级。

升级模拟器和核心 application

要想升级模拟器和核心 application，你只需在新版本的发行包文件中提供它们的新版本即可。在生成 *relup* 文件的同时，其中便已经为你做好了一切。你只需记住在升级 Erlang 运行时系统时调用的 `systools:make_tar/2` 中要包含 `erts` 选项，因为这样才会在新的 tar 文件中包含模拟器。这听起来很简单，确实是，但也有一些需要留意的东西。

升级模拟器和核心 application（包括 *erts*、*kernel*、*stdlib* 和 *sasl*）需要重启虚拟机，这通常由 `restart_new_emulator` 指令触发。与其他升级操作不同，这将是文件中执行的第一条指令，它令重启后的模拟器和核心 application 为新版本，但普通 application 仍使用旧版本。这种两阶段方法允许其他升级的 *behavior* 和特殊进程能在调用 `code_change` 的过程中使用新版本的核心 application。

如果你对此方法不满意，可以手工编辑 *relup* 文件。使用 `restart_emulator` 指令替换 `restart_new_emulator` 指令，这样重启模拟器时将统一使用新版本的 application。当你在 *relup* 文件中使用 `restart_emulator`（非 `restart_new_emulator`）重启模拟器时，它应当是执行的最后一条指令，因为它所做的操作仅仅是使用新的引导文件重新启动系统。这意味着位于 `restart_emulator` 之后的指令都会被忽略，而任何在它之前的指令都是使用旧的模拟器执行的。除此之外，还可以手工添加 `apply` 指令，它很有用，如果希望直接启动新发行包的 application，则可以使用该指令，而非 `code_change`。

非向后兼容的升级和降级

有的时候，你使用了 `restart_new_emulator` 指令来重启，这样一来，重启后模拟器和核心 application 都更新为新版了，但你的 application 还是老版本的，这在你的预料之中，因为你打算稍后再更新它们。但如果升级过程跨越多个发行包版本，那么你可能会面临一个问题，即你的非核心 application 中所调用的某些核心 application 中提供的函数已被弃用了。一般来说，标记为已弃用（deprecated）的函数会继续跨两个主发行包版本保留，但编译任何使用它们的代码时都会打印出警告，使得你有机会安全地删除这些函数调用。解决方案是尽快更换任何弃用的功能，并在测试升级时分几个步骤进行升级，以确保所有 application 都是向前兼容的。

353

如果你仍在运行比 R15 更老版本的发行包（我知道有不少这种情况），那么降级时可能会遇到问题，因为当降级时重启老的模拟器后，在系统尝试加载 beam 文件时，由于其格式不兼容将引发问题。如果你受此影响，请通过使用旧模拟器及其相应发行包的编译器编译新代码来解决。

在这两种特殊情况下，对升级和降级执行测试变得至关重要，测试可尽可能早地让你发现潜在的问题和不兼容。

使用 rebar3 进行升级

现在你已了解与升级有关的所有细节了，让我们看看如何使用第 11 章的“rebar3”一节中介绍过的 `rebar3` 工具来完成升级吧。首先，使用 `rebar3` 来构建一个发行包，再从 `coffee-1.0` 启动。所需的命令与我们在“rebar3”一节中使用的类似：

```
$ mkdir ernie
$ cd ernie2
$ rebar3 new release coffee desc="Coffee Machine Controller"
$ cd coffee
$ perl -i -pe 's/0\.1\.0/1.0/' ./apps/coffee/src/coffee.app.src ./rebar.config
$ cp <path-to-coffee-1.0>/coffee-1.0/src/*.erl apps/coffee/src
$ rebar3 as prod compile
==> Verifying dependencies...
==> Compiling coffee
_build/default/lib/coffee/src/coffee_fsm.erl:2:
  Warning: undefined callback function code_change/4 (behaviour 'gen_fsm')
_build/default/lib/coffee/src/coffee_fsm.erl:2:
  Warning: undefined callback function handle_event/3 (behaviour 'gen_fsm')
_build/default/lib/coffee/src/coffee_fsm.erl:2:
  Warning: undefined callback function handle_info/3 (behaviour 'gen_fsm')
_build/default/lib/coffee/src/coffee_fsm.erl:2:
```

```

Warning: undefined callback function handle_sync_event/4 (behaviour 'gen_fsm')
$ rebar3 as prod release
==> Verifying dependencies...
==> Compiling coffee
...<snip>...
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    /Users/francescoc/ernie2/coffee/_build/prod/lib
    /Users/francescoc/ernie2/coffee/apps
    /usr/local/lib/erlang/lib
    /Users/francescoc/ernie2/coffee/_build/prod/rel
==> Resolved coffee-1.0
==> Including Erts from /usr/local/lib/erlang
==> release successfully created!

```

使用 *rebar3* 发行包模板为我们的 *coffee* application 建立了一个目录结构，然后将版本号更改为 1.0，将 *coffee-1.0* 的源代码复制到新发行包目录中，运行 *rebar3 compile* 以验证代码是否有效（正如我们先前所见，编译 *coffee_fsm.erl* 时出现了警告信息，原因是缺少一个回调函数），然后使用 *prod* 配置文件（profile）构建了发行包。现在我们可以启动发行包以确定它是否能正确运行：

```

$ ./_build/prod/rel/coffee/bin/coffee console
...<snip>...
Machine:Rebooted Hardware
Display:Make Your Selection

=PROGRESS REPORT==== 24-Jan-2016::16:06:10 ===
    supervisor: {local,sasl_safe_sup}
        started: [{pid,<0.213.0>},
                    {id,alarm_handler},
                    {mfargs,{alarm_handler,start_link,[]}},
                    {restart_type,permanent},
                    {shutdown,2000},
                    {child_type,worker}]
...<snip>...
=PROGRESS REPORT==== 24-Jan-2016::16:06:10 ===
    application: sasl
        started_at: coffee@francescoc
Eshell V7.2 (abort with ^G)
(coffee@francescoc)1> application:which_applications().
[{sasl,"SASL CXC 138 11","2.6.1"},
 {coffee,"Coffee Machine Controller","1.0"},
 {stdlib,"ERTS CXC 138 10","2.7"},
 {kernel,"ERTS CXC 138 10","4.1.1"}]

```


于是我们得到了版本号为 1.0 的 *coffee* 发行包。接下来，我们需要制作 1.1 版的发行包，因此将该发行包的 *coffee_fsm.erl* 复制到源代码目录中，升高了版本号，然后生成新发行包：

```
$ cp <path-to-coffee-1.1>/coffee-1.1/src/coffee_fsm.erl apps/coffee/src
$ perl -i -pe 's/1\.0/1.1/' ./apps/coffee/src/coffee.app.src ./rebar.config
$ rebar3 as prod release
==> Verifying dependencies...
==> Compiling coffee
...<snip>...
==> Resolved coffee-1.1
==> Including Erts from /usr/local/lib/erlang
==> release successfully created!
```

在生成 *relup* 文件之前，需要先准备好 *coffee.appup* 文件。因为 *rebar3* 不会在常规位置创建 *ebin* 目录，所以我们自己创建该目录，并将 *coffee.appup* 文件复制到哪里，然后使用 *rebar3 relup* 命令：

```
$ mkdir apps/coffee/ebin
$ cp <path-to-coffee-1.1>/coffee-1.1/ebin/coffee.appup apps/coffee/ebin
$ rebar3 as prod relup
==> Verifying dependencies...
==> Compiling coffee
==> Starting relx build process ...
...<snip>...
==> Resolved coffee-1.1
==> Including Erts from /usr/local/lib/erlang
==> release successfully created!
==> Starting relx build process ...
...<snip>...
==> Resolved coffee-1.1
==> relup successfully created!
```

如果我们查看生成的 *relup* 文件的内容，会发现它与本章前面“发行包升级文件”一节中使用 *systools:make_relup/4* 生成的内容相同：

```
$ cat ./_build/prod/rel/coffee/relup
{"1.1",
 [{ "1.0", [],
   [{load_object_code,{coffee,"1.1"},[coffee_fsm]}},
    point_of_no_return,
    {suspend,[coffee_fsm]},
    {load,{coffee_fsm,brutal_purge,brutal_purge}},
    {code_change,up,[{coffee_fsm,{}}]}],
```

```

    {resume,[coffee_fsm]]}],
[{"1.0",[],
 [{load_object_code,{coffee,"1.0"},[coffee_fsm]}},
  point_of_no_return,
  {suspend,[coffee_fsm]},
  {code_change,down,[{coffee_fsm,{}]}},
  {load,{coffee_fsm,brutal_purge,brutal_purge}},
  {resume,[coffee_fsm]]}]]}.

```

现在,你可以通过 `rebar3 as prod tar` 创建一个压缩包了,并按照本章前面“安装升级”一节中所介绍的方法进行安装和升级。

假设你使用 *rebar3* 作为构建和发布工具,那么值得花一点时间看看由 Richard Jones 编写的 *reflow* 工具 (<https://github.com/RJ/reflow>)。如果你使用 Git 进行版本控制,并使用 *rebar3* 做发行包生成和升级,*reflow* 就是为你而设计的,它的目标是解决升级过程中各种烦琐的操作,例如升高版本号 and 创建 *.appup* 文件等。

356

总结

就像我们在本书中看到的大多数事情一样,Erlang 提供了强大的基本语言结构,OTP 基于此构建出的库和框架隐藏了复杂性,简化了基于 Erlang 的系统的开发、部署和维护。我们从处理运行时系统模块加载的 `code:load_file/1` 开始,明白了如何管理进程状态的变更、数据库模式的变更、进程及其依赖关系的同步,以及分布式环境中的依赖问题。

为了升级目标系统,你需要从基准(baseline)安装开始。它通常是第一个发行包,并且是由你手工创建的。除非你使用 *rebar3*,否则这些工作必然是手工完成的,因为大多数发行包升级工具都是用 Erlang 编写的,倘若没有基准系统,你便无法运行它们。可见这是一个经典的鸡与蛋问题。

基准发行包准备就绪后,你需要按照以下步骤达成系统升级。不要害怕,因为这些步骤中大多数要么是自动的,要么可由现有的工具来处理,或者两者都是:

- 添加新功能,将其打包到相应的模块和 application 中,并升高模块和 application 的版本。
- 创建新 rel 文件,其中包含新的 application 和要升级的 application,同时省略打算删除的 application。
- 生成启动脚本和新的 *sys.config* 文件,并确保能用它们启动新发行包。
- 如果作为升级的一部分有 behavior 或特殊进程需要做状态更改,或使用不同的数据格式(包括数据库模式更改),请将状态和数据格式从旧发行包传递到新发行

包的 `code_change` 函数并返回。

- 为每个要升级的 application 编写一个 `.appup` 文件。将这些文件放在 `ebin` 目录中。
- 创建一个 `relup` 文件，其中包含升级过程中需要执行的低级指令。
- 创建一个包，并把它放在（当前安装的）目标发行包的 `releases` 目录中。
- 解压缩并安装它。
- 如果一切稳定，让你的新发行包永久化。如果发现不稳定，请重新启动节点，进而重新启用旧版发行包。

发行包解压后，当前被升级的节点上将会发生一系列的转换。当你安装发行包并且升级成功时，系统便开始启动并运行新版发行包。如果升级操作因某种原因失败，系统将重新启动并恢复到以前的版本。运行新发行包的过程中，可以将其永久化。永久化后，之后再重新启动节点仍会重新启动最新版发行包（参见图 12-6）。

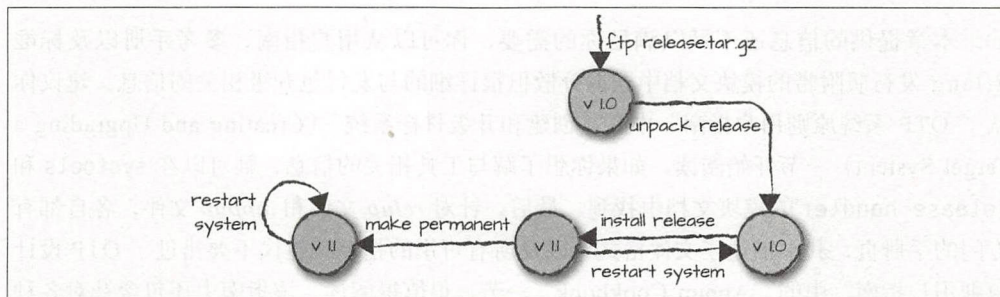


图 12-6: 升级发行包

我们还介绍了分布式环境中的升级，这些升级过程允许你同步节点。这种状况确实会发生在现实世界中，但只适用于网络可靠的非常小的集群。如果你正在处理分布式数据中心、云计算、虚拟化以及许多其他层次的复杂性和不稳定性，则需要采取不同的升级方案。确保旧节点和新节点之间是向后兼容并可互操作的，满足这一点使它们能够共存于同一集群中。先升级少量的几个节点，并在此过程中监控它们以确保一切正常，然后陆续升级更多节点。如果你丢失了一些机器或者遇到网络分区或其他升级故障，请继续尝试，直到所有节点都成功升级。

让我们做更进一步的探讨。对没有单点故障并且有多个节点正在运行的集群而言，在线升级是否真的有意义？如果你能够进行滚动升级（rolling upgrade），能在不丢失任何请求和中断任何流量的情况下干净地关闭节点，那么是否一次关闭一个节点，升级其代码，然后重新启动它以将其重新引入集群实际上更容易？你能升级你的代码，而不会显示大多数在线银行向我们频繁显示过的令人尴尬的“系统因维护而关闭，如有需要请联系我们，采取此做法是因为您的业务非常重要，我们非常重视您作为客户的权益”提示，并

确保你不会因升级而丢失任何请求。

采用何种升级方案，需要考虑集群大小，已有基础设施，系统冗余容量以及团队的经验 and 规模。软件升级需要花费时间和金钱来实现、测试和部署。倘若出现问题——事实上大多数升级过程都会出一些问题——如果你是一家不需要提供 99.999% 可用性的创业公司，自然无人会在乎你是否偶尔反复升降级你的节点。但是，如果你要升级的是数以万计的交换机，每台交换机处理数百万用户流量，并且停机和中断需承担违约损失，亦或者要升级的是每分钟都会产生数千美元收入的电子商务网站，那么用户将会很在意你的升级！

软件升级是一个独特而强大的功能，你可以在罕见但关键的时刻使用它。只要有价值，即使需要一些额外的工作也值得使用，记得在重负载下测试升级和降级过程，并尽量覆盖尽可能多的故障情况。

如果本章提供的信息还不足以满足你的需要，你可以从用户指南、参考手册以及标准 Erlang 发行版附带的模块文档中获取分散但很详细的与发行包升级相关的信息。建议你从“OTP 系统原则用户指南”中的“创建和升级目标系统”（Creating and Upgrading a Target System）一节开始阅读。如果你想了解与工具相关的信息，则可以在 `systools` 和 `release_handler` 的模块文档中找到。最后，针对 `relup` 文件和 `.appup` 文件，各自都有专门的手册页，其中描述了文件格式，以及所有可用的指令。建议不要错过“OTP 设计原则用户指南”中的“Appup Cookbook”一节，很值得阅读。该指南中还包含针对各种 `behavior` 和特殊进程的 `code_change` 函数的说明。

尽管如此，正如我们在第 11 章中建议过的：理解底层概念、工具和程序非常重要，但除非项目有特殊需求，否则最好使用 `rebar3`。它能处理各种与发行包制作和升级相关的烦琐任务，并允许你在必要时进行扩展，同时提供了良好的社区支持，当你需要建议或协助时，这很有用。

接下来是什么

基于本章提供的知识，你已经了解了如何打包发行包并在不影响流量的情况下执行在线升级，现在是时候考虑系统架构了。如果你想要一个五个九级别可用性的系统，各个生产节点应该具备哪些基本功能？为了实现节点可伸缩性，应该采用哪种分布式体系结构模式？在下一章中，我们将揭晓答案。你还在等什么？翻开下一页赶紧阅读吧！

分布式架构

第 12 章已经介绍了单个简单节点的实现。节点是最小的独立可执行单元，其上运行着 Erlang 运行时系统。在本章，我们开始展示如何从单个节点扩展为由多个节点构成的分布式系统。我们会介绍如何通过这些节点实现可用性、可伸缩性和一致性。这些品质与可靠性一起保障了你的系统行为，使其即使在异常情况（如出故障或极端负载）下也能正常运行。

每个节点由一系列低耦合的 OTP application 组成，定义于 OTP release 文件中。节点提供的服务，以及它能够处理的任务都是由 OTP release 决定的。共享相同 release 文件的节点包含的 OTP application 相同，因而这些节点被认为属于同一类型。

在集群中，一种类型的节点可以与其他类型的节点交互，提供系统的端到端功能。对一个 Erlang 系统来说，只包含一个节点尽管可行，但更常见的情况是由许多节点构成，它们构成一个或者若干个集群。

我们之所以需要集群，原因有很多。比如当实现微服务架构时，可以通过每个集群提供一组服务。或者你可以利用集群来实现可伸缩性，对集群分片，以提高计算能力和可用性。当面对那些涉及多个数据中心并且各自地理位置分散、运行环境不统一的分布式 Erlang 系统时，没有一套能够普遍适用的方案，需要有针对性地进行设计。这也就意味着那些提供监视、管理、编排 Erlang 节点的工具和框架都必须能够满足不同集群模式的需要。当部署到 Amazon 或 Rackspace 时，一些工具可能很适合，但是当换成 Parallela 或者 Raspberry Pi 集群就不行了。其他有些工具则是部署到裸机时很好用，换成虚拟环境就又不行了。

在本章和接下来的几章中，我们将会带你入门分布式架构设计。本章先介绍 Erlang 节点类型这一概念，然后描述如何把它们分组，以及如何交互。这可以帮助你确定如何把系统分割为独立的节点，让每个节点提供特定的服务。我们描述了用于提供这些服务

的最常用的分布式架构模式，以及一些流行的框架，比如 Riak Core 和可伸缩的分布式 Erlang。

虽然对于 Erlang 来说分布式功能是开箱即用的，但是对你的工作而言它们并不一定是最合适的工具。我们还介绍了你可能会用到的一些方法，用于把 Erlang 与非 Erlang 节点相互连接。最后我们总结了一套高级的方法，可以用于定义各个节点类型的接口与数据模型。

节点类型与家族

直到最近，都还没有涵盖分布式 Erlang 系统的通行定义。OTP 在定义单个节点的各个组件方面做了很棒的工作，但是止步于此，没有提供节点分组以及在集群里交互方面的描述。尽管当世界各地的开发人员谈到 generic servers、applications 和 releases 时已经不会有歧义，但是当讨论到集群、节点在集群中的角色，或者伸缩模式时，还是存在一些混淆。后来，在 RELEASE（一个由欧盟资助的旨在研究分布式多核架构下 Erlang VM 伸缩性问题的项目）里，这些概念得到了讨论和规范化。^{注1} 在开始讨论分布式架构之前，让我们先定义清楚术语。

想象这样一个由三个 Erlang 节点构成的系统。第一个节点运行着 Web 服务，与客户端维持着大量 TCP 连接。客户端是移动 App 或者 Web 浏览器。它接收 HTTP 请求，解析为 Erlang 项，然后转发给第二个节点，即处理系统业务逻辑的节点。

在处理请求时，第二个节点可能会与另外的节点交互，该节点也提供某种服务。为了简单起见，我们假设那是一个数据库节点，有可能（但并不必须）是用 Erlang 编写的。从终端用户的角度来看，这一切只不过是一个以黑盒方式访问的单一系统而已。对于用户来说，Erlang、节点、节点间的分布式层等都是看不见的。

图 13-1 所示的例子有三种语义节点类型，分类的依据是节点在集群中的功能和用途。

361 同一类型的多个节点实例可能运行的是同一发行包的不同版本。我们之所以会运行单个节点的多个实例，是为了可用性和可伸缩性。在第 14 章和第 15 章我们会更详细地讨论这些主题。

我们把 Web 服务器节点称为前端节点 (front-end node)。前端节点负责与外部客户端相连，并处理所有传入的请求。它们的角色相当于网关 (gateway)，根据需保持与客户端的连接，格式化入站请求和出站响应，并把请求转给那些负责处理业务逻辑的节点。它们是服务端软件的一部分，为表现 (presentation) 层提供服务，但并不直接承载表现层。

注 1 “Distributed Erlang Component Ontology,” 30 June 2013 by Hoffmann, Cesarini, Fernandez, Thompson & Chechina.

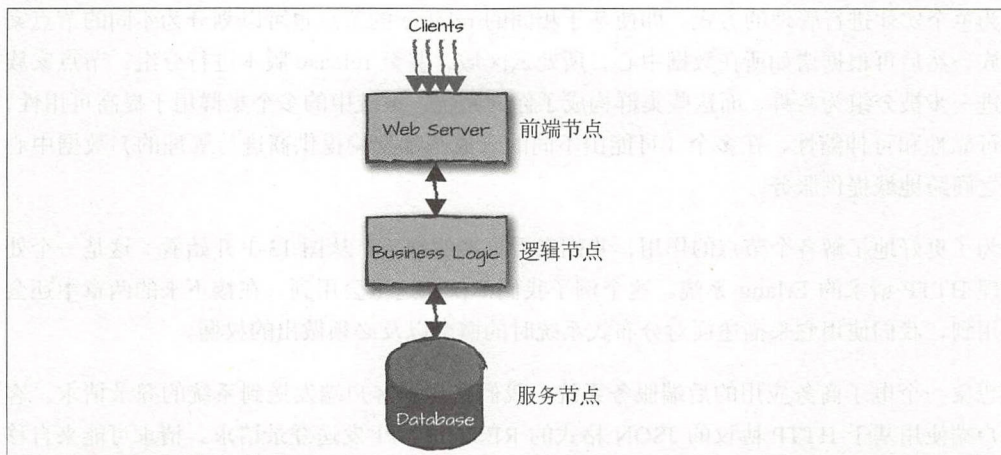


图13-1: 语义节点类型

逻辑节点 (logic node, 通常也称其为后端节点) 实现了系统的业务逻辑。它们包含处理由前端节点转发来的客户端请求所需的所有代码。并且当处理请求时, 它们还能够缓存会话数据和访问其他节点上的外部服务。

最后, 我们还拥有服务节点 (service node), 为逻辑节点提供服务。例如数据库服务、认证服务或支付网关服务等。服务节点本身还可以提供与第三方服务和 API 的连接。

节点类型只是我们描述各个节点的大致责任的一种方式。在小型或简单系统中, 单个节点可以同时具备多种职责, 例如既是前端节点也是逻辑节点, 甚至也是服务节点, 三合一。试想这样一个节点, 其上运行着 Erlang Web 服务器 (如 Yaws、Webmachine、Cowboy)、Erlang/OTP 胶水和业务逻辑, 以及 Erlang 数据库 (如 Mnesia、CouchDB、Riak), 并且这一切都在同一个虚拟机内。把这些 application 组合到单个节点后, 由于运行在相同的内存空间内, 减少了跨节点的 I/O 以及网络开销, 但是也产生了单点故障以及可能无法伸缩的架构。相比之下, 在多节点系统中, 不同类型的节点承担着不同的职责, 实现了可维护性、可伸缩性和可用性。

362

在你按照功能划分节点类型时, 尝试把内存密集的和 CPU 密集的功能划分到单独的节点中。这有助于虚拟机调优, 并使你能灵活地选择底层硬件, 优化成本和性能。这还有助于你最小化系统故障的风险, 因为简单的节点更容易实现和测试, 而且当它们出现故障时, 不会影响到其他节点。蜂拥而至的并发请求导致一个节点的内存耗尽不会影响到用户数据库或者客户端连接。(我们会在第 15 章的“负载调节与背压”一节讨论如何处理这种情况。)

我们把运行相同 OTP release 的节点类型划分为同一节点家族。这是一种将多个节点作

为单个实体进行管理的方式。即使基于相同的 release 的节点也可以划分为不同的节点家族，然后再根据诸如所在数据中心、所处云区域，甚至 release 版本进行分组。节点家族进一步被分组为集群，而这些集群构成了你的系统。系统中的多个集群用于提高可用性、可靠性和可伸缩性，在多个（可能由不同的云或基础架构提供商进行管理的）数据中心之间跨地域提供服务。

为了更好地了解各个节点的作用，我们来看更多的细节。从图 13-1 开始看：这是一个处理 HTTP 请求的 Erlang 系统。这个例子我们不仅在这里会用到，在接下来的两章中还会用到，我们使用它来描述面对分布式系统时的概念以及必须做出的权衡。

想象一个电子商务应用的后端服务系统。我们聚焦于客户端发送到系统的登录请求。客户端使用基于 HTTP 协议的 JSON 格式的 RESTful API 发送登录请求。请求可能来自移动 App 或 Web 浏览器。请求由在前端节点上运行的 Web 服务器接收，该服务器将其解析为 Erlang 项并将其转发到逻辑节点。转发的项包括登录请求、用户 ID 和加密过的密码。

逻辑节点首先检查请求的有效性，然后通过认证服务器认证用户。如果成功，逻辑节点将在本节点内创建会话 ID 与记录 (record)，并缓存。然后把会话 ID 返回给前端服务器，前端服务器进行编码后返回给客户端，确认登录请求已成功。在会话期间，客户端在所有后续通信中都会使用获得的会话 ID，在每个后续请求中，将该 ID 传递给逻辑节点用于检索对应的记录。

无论你使用的是和图 13-1 类似的三层架构还是使用其他架构模式，逻辑节点都是重要的中介和检查点。要避免前端节点直接与服务节点通信，这样做当然说不上非法，但是会让系统结构变得糟糕，当别人尝试从架构层面了解系统时会感到困惑。

我们在架构中添加了多个节点类型的实例来创建分布式集群模式，也称为系统蓝图。如果你只是想要一个静态的架构，那么扩展规模只需添加一些独立的（即相互之间没有交互的）系统实例就已足够，这种系统蓝图很简单。但是，如果你的应用程序是一个全球性在线商店，服务器需要分布在各个国家，希望能够根据访问来源 IP 的地理位置自动路由匹配，并具有高峰和低谷期的动态伸缩能力——在发薪日 (payday)、黑色星期五和圣诞节等来临时弹性地增加计算资源，多至能允许扩展到支持百万用户同时在线，每秒十万次请求处理，然后当不再需要时缓缓释放，那么你就从系统开始设计之初就得额外考虑很多事情了。

不论对于静态架构还是动态架构，集群中的节点（和硬件）的管理方式都与你选择的跨节点、跨节点家族、跨集群的数据分布策略紧密相关。你采用何种方式将节点和集群连接在一起，以及在其中采用何种策略进行数据复制都是重要的问题。用户登录到系统并购物。你是在所有节点上都保存用户会话数据的副本还是只在部分节点上保持？每当客

户将商品添加到购物车中时，如何把变化传播给其他节点？如果有网络分区或网络故障会发生什么？软件错误或者节点终止又会怎样？我们将在第 14 章介绍这些设计选择。这一切问题最终都会归结为在可用性、可靠性、一致性和可扩展性之间的取舍问题。你需要早早做出的是了解适合你正在架构的系统的需求和你要提供的最终用户体验的妥协。

联网

到目前为止，我们谈论的是前端节点与逻辑节点进行通信，逻辑节点又将请求发送到服务节点。我们没有提到分布式 Erlang，是因为它虽然是同一数据中心内较小集群的理想选择，但是当跨多个数据中心，以及考虑到安全性、可用性和大规模的可伸缩性时，它就未必是正确的解决方案了。在某些情况下，当需要传输大量数据时，单个套接字将成为瓶颈，这时你可能希望使用库（例如 ranch 或 poolboy）中提供的连接池。RESTful API 为你提供了平台独立性，其他协议（如 AMQP、SNMP、MQTT 和 XMPP）也是。分布式 Erlang 可能仍然符合你的需求，但相较于使用 TCP，你可能希望使用其他替代品，如 0MQ、UDP、SSL 或 MPI 等。

在某些系统中，网络拓扑将会为不同类型的流量提供不同的网络。流量监控、计费、配置和维护相关的流量走的是操作和运维网络（O&M 网络），而其他流量如建立呼叫、即时消息、短信或遥测数据等则会走数据网络。之所以把它们分开，是因为数据网络相比 O&M 网络拥有更高的带宽和可用性。对于大规模多用户在线游戏的用户，你要避免停止或拖慢它们，但对于移动和处理结算记录的过程却可以有一些延迟。

< 364

通过示例中展示的决策网络有助于你弄清楚必须做出的选择。如果你担心电子商务网站的安全性，你可能希望将前端节点放置在非军事区域（DMZ）中，也称为外围网络（参见图 13-2）。它是网络的一个物理或逻辑部分，用于将你的节点暴露给不可信的网络（例如互联网），用户访问你的服务经由的正是不可信的网络。传统的 DMZ 是实现在硬件中的，结合了网络元素的安排管理，以及使用防火墙软件和其他安全措施等。在云计算环境中，没有硬件组件能让你用，只能通过网络连接和防火墙规则来模拟。不过最终结果仍然是一样的。通过在后端节点周围创建一个额外的安全层，你可以通过不暴露其接口降低逻辑节点和服务节点遭入侵的风险。

如果你使用分布式 Erlang，能够访问前端节点很大程度上也就意味着能够访问你的逻辑节点和服务节点。而如果这时候相关人员对 Erlang 缺乏常识，以及对于安全性一知半解，那么系统安全性就堪忧了。事实上在这种情况下，你必须在 Web 服务器和运行业务逻辑的节点之间使用套接字，甚至是加密套接字通信，对每个请求都要进行身份验证并检查其有效性。但是对于业务逻辑节点与数据库节点，由于它们位于安全的防火墙之后，因此这些节点间可以使用分布式 Erlang 透明地进行通信。

< 365

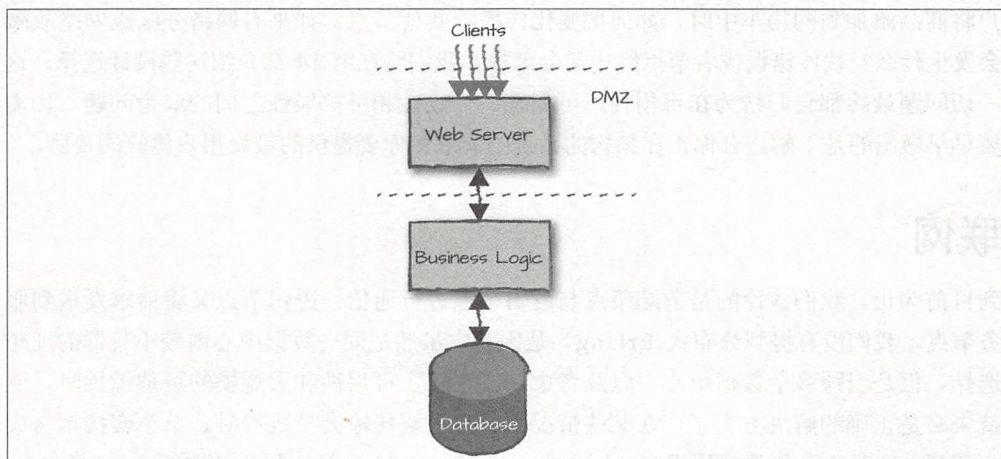


图13-2: 非军事区

分布式计算误区

如果你认为你的网络是可靠的, 极少有网络分区, 那请你再想想! 网络问题往往在你最意想不到的时候出现, 并且如果你没有处理所有可能的边界情况, 其副作用及后果可能是灾难性的。这些早在几十年前就被 Peter Deutsch 和他的同事在 *fallacies of distributed computing* (分布式计算误区) 一文中提到过了, 但依然与我们今天设计的系统紧密相关。

如果与远程节点的连接断开或者拥塞了, 你如何确定这是一个网络问题, 而不是目标节点崩溃或者响应慢? 你是通过发回的错误信息判断 (当然前提是发送了这样的信息), 还是在另一个节点上重试相同的调用来判断? 假设你用后一种方法, 那你需要考虑到重复请求可能引发的副作用。事实上要区分到底是节点崩溃还是节点缓慢是不可能的。尽管如此, 最好还是梳理清楚各种请求及其处理过程中可能伴随产生的错误。

除此之外你还要记住, 跨节点操作的 CPU 和 I/O 开销比直接在本地执行要高, 原因是序列化操作、网络接口虚拟化、协议处理这些都会带来额外开销。带宽不是无限的, 网络延迟会影响请求的端到端性能。当你划分负载、压测系统时, 要把这些记在心中, 才能更好地调优。这些问题并不只是软件问题, 你的硬件系统和基础设施也同样关键。随着系统的运行, 新的节点 / 计算机会加入, 同样也会离开, 在你的程序中需要高效地处理因此而带来的网络拓扑结构的改变。

最后, 别忘了考虑你那友善的系统管理员, 他们也难免有时会不按流程、犯错、

忽视警告。*fallacies of distributed computing* 一文中指出，同一网络的系统管理员甚至可能分属不同的组织机构。这带来的风险不仅仅是错接网线、搞乱配置，或者是对拓扑结构管理策略持有相左的意见。还挑战着你的软件如何看待并处理现实世界的各种状况。你的软件能够处理——由于负载均衡器 / 防火墙代码问题或配置错误导致的——两倍的流量吗？

如果你用的是云基础设施，不清楚其拓扑结构，那么实现弹性就更难了，因为这些环境下的分区问题更难理解和排查。云计算环境下实例的运行速度通常更慢，而网络通常更繁忙，都让这一目标更难达到。

分布式 Erlang

366

使用分布式 Erlang 实现你的架构有两种途径。一种方式是静态集群——具有固定数量的已知参数，并且具有固定的标识（主机名、IP、MAC 地址等），不提供动态伸缩。另一种方式则是动态集群——标识和节点的数量可以在运行过程中变化。不管是哪一种情况，你的系统都需要实现传递性连接（transitive connection），因为网络连接、节点自身都可能会出故障（并重启）。静态和动态系统之间的唯一区别是，在后者中，除了故障之外，节点的启动和停止是以更可控的方式进行的。在静态系统中，除非故障，否则节点不会停止运行。

全连接的分布式 Erlang 集群（参见图 13-3）对于某些大小和要求的系统是理想的，但正如我们以前说过的，没有普适性的解决方案。截止到本书写作的时候，根据你的节点配置以及跨节点发送的消息的大小和频率，完全网格化的 Erlang 集群规模扩大到 70 到 100 个节点后，性能开始明显下降。当将一个新节点添加到集群中时，与其共享相同秘密 cookie 的所有可见（非隐藏）节点的信息都会传播给它，并相互连接，监视掉线。因此，如果有 100 个相互连接的节点，就会产生 5050 个 TCP 连接（ $100 + 99 + \dots + 2 + 1$ ）和所有这些节点之间的心跳，从而产生了节点和网络上的开销。其他的单进程瓶颈也存在，例如处理 Erlang 远程过程调用（RPC）的 *rex*，或者是远程分裂进程并处理网络监控的网络核心部分（net kernel）。

能够把全连接的分布式 Erlang 集群的规模扩大到什么程度取决于系统具备的特点。我们在第 2 章的“节点间的连接与可见性”一节介绍的隐藏节点，其作用正是充当网关，阻止信息跨越全连接节点集群的传播。它们为你提供了隔离和可伸缩性，但是你必须要在它们之上打造相应的框架。你应该看看其他替代方案或现有框架，例如 Riak Core 和 SD Erlang 等，这些框架将在接下来的小节中进行介绍。

最终，你可以创建一个使用 SSL 而不是简单 TCP 来承载的特殊的 Erlang 发行包。你可以在“安全套接字层用户指南”的“Using SSL for Erlang Distribution”部分（<http://bit.ly/erlang-ssl>）找到更多可阅读的信息。

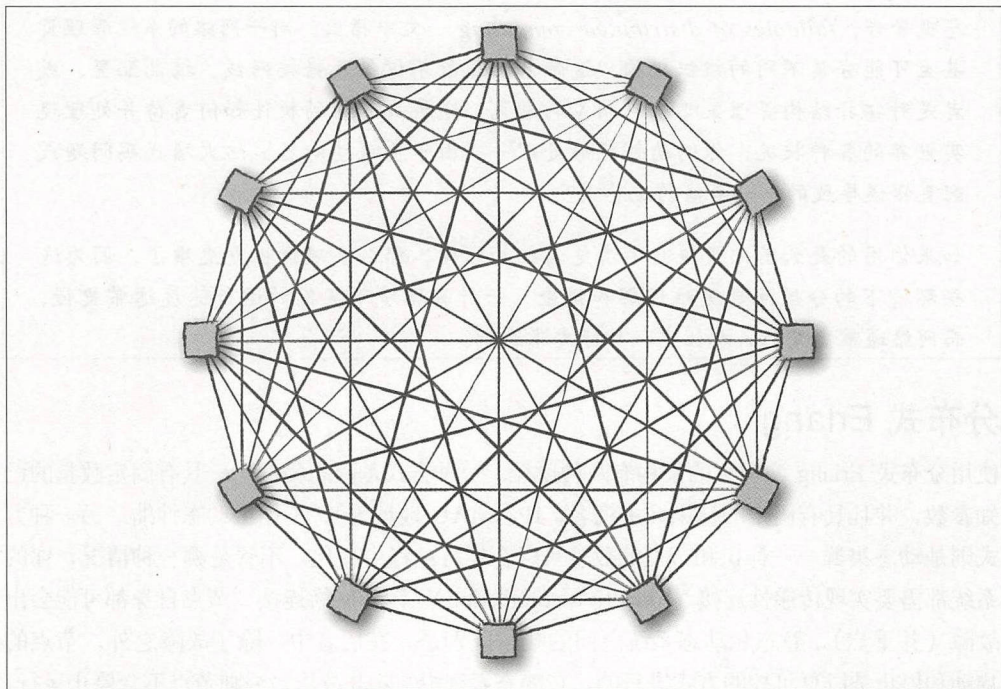


图13-3: 分布式 Erlang

367



使用进程 PID

如果你在分布式 Erlang 集群中使用进程 pid 而非注册过的名字, 请记住, 如果远程节点崩溃并重启, 该 pid 在重启后的节点上可能会被其他进程重用。这可能导致你发给该 pid 的消息并不像你想象的那样由正确的目标进程接收, 而是发给了一个预料之外的进程。在这类场景下, 你应当永远保持对远程节点和进程的检视, 并在检测到故障时采取适当的措施。有一个跨节点的进程 ID 计数器, 它为你提供了重启时重用相同 pid 的服务, 从而避免了此类问题, 但前提是 Erlang 端口映射守护程序 (epmd) 必须持续运行。

Riak Core

Riak Core 是一种在无主节点系统上实现了最终一致性数据复制模型的框架, 提供了高可用性, 并保证无单点故障。它是建立在分布式 Erlang 之上的, 并且是分布式 Riak 键值存储的基础, 基于的是 Amazon 2007 Dynamo 论文 (<http://bit.ly/riak-dynamo>) 中的想法。对于需要高可用性, 以及节点和网络出故障后能够自愈的系统, 可以说它是理想的框架。

完全解释 Riak Core 的所有细节需要另写一本书了, 因此我们在此只介绍它在分布式框架领域能够成为有力竞争者的亮点部分。

Riak Core 运行在物理节点集群上, 然后又在此基础上建立了一套虚拟的节点系统, 称为

`vnode`。`vnode` 的数量是可配置的,一般来说典型的 Riak Core 集群包含 15~20 个物理节点,在此基础上共同承载 256 个 `vnode`。每个 `vnode` 占据一块 SHA-1 散列函数定义的 160 比特的整数区间, Riak Core 把这作为其一致性哈希系统的基础。一致性哈希能把键值数据均匀分布在整个集群中,并且当在集群中添加或移除物理节点时还最小化了需要重定位 (relocation) 的数据量。

要将数据存储 in Riak Core 集群中,客户端会发送一个写入请求,其中包含键和值。Riak Core 计算出键的哈希值,然后确定该值到底落在哪个 `vnode` 所拥有的 160 比特值区间内。Riak Core 会复制写入操作,因此它首先确定写入请求的复制因子 (replication factor),称为 N ,并且默认值通常为 3。然后它存储数据的 N 个副本,在主 `vnode` 中存一份,其余的存储在哈希空间相连的下 $N-1$ 个 `vnode` 上。当写入的副本数量等于写入因子 (write factor) W 时, Riak Core 就认为写入完成了。默认情况下, W 为 $N/2+1$,但如果 N 为 3,则 W 为 2。

要从集群读取数据,客户端发送一条请求,其中包含键名。Riak Core 首先计算出键的哈希值,以确定保存了请求的值的主 `vnode` 是哪一个。然后请求该 `vnode` 的值和连续的下 $N-1$ 个 `vnode`,并等待,直至达到被称为 R 的读取因子 (read factor)。像 W 一样,默认情况下 R 为 $N/2+1$,而如果 N 为 3,则 R 为 2。一旦成功读取到值的两个副本, Riak Core 就将向客户端返回请求的值。

当 Riak Core 集群首次被创建时,其物理节点会声明对 `vnode` 的所有权,这样做的目的是使相邻的 `vnode` 不要位于同一物理节点上。这样一来,通过将副本存储在毗邻的 `vnode` 中,并且假设该集群至少包括最少推荐的 5 个物理节点, Riak Core 能够保证副本可以尽可能存储在不同的节点上。如果任何物理节点崩溃或变得不可达,则其他副本仍然可以响应读取或写入该数据的请求,从而在集群被分区的情况下依然能提供高可用性。图 13-4 清楚地展示了虚拟节点在物理节点上的分布,当查找值时,键的哈希值指明了 `vnode`,后者又指向负责该值的主 Erlang 节点。

使用 `vnode` 和一致性哈希的一个优点与当节点被添加或停止服务时发生的重组 (reshuffling) 有关。假设图 13-4 所示的集群里有 16 个节点,然后使节点 1 永久性停止服务。Riak Core 只需在现有节点上重新分布 `vnode` 1、17、33 和 49,而无须在所有节点上重组所有数据。仍在服务中的物理节点上的 `vnode` 保持不变。如果加入一个新节点,则 4 个 `vnode` 将从当前位置移动到它,仅影响 `vnode` 所在的节点。

Riak Core 中的节点是对等的,没有主节点。节点使用 *gossip* 协议将共享信息 (如集群拓扑变化、`vnode` 声明等) 传送给其他随机选择的节点。如果由于某种原因部分节点错过了集群拓扑更新通知,那么 *gossip* 协议将自动弥补,确保系统自愈。

< 369

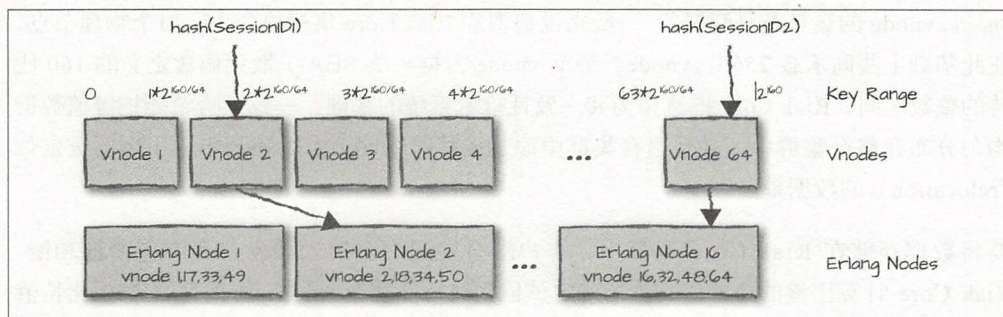


图 13-4: 虚拟节点的vnode

Riak Core 使用了 *hinted handoffs* (提示再转手) 机制, 该机制的作用是即使核心 (primary) vnode 或者部分复制 (replica) vnode 由于网络分区而下线或不可达, 也能保证存储一份数据维持多份副本。在此类故障情况下, Riak Core 把数据存在备选 (alternative) vnode 中, 并给这些节点一个提示 (hint), 指明数据真正应当存储的位置。之后当不可达的那些 vnode 恢复正常后, alternative vnode 把暂存的数据转手给这些 vnode, 从而修复了系统。这种 *hinted handoffs* 机制是 Riak Core 的 *sloppy quorums* (松散仲裁) 机制的一部分。Riak 要求写入操作要被视为成功必须有 W 个确认才行, 而简单的读取操作则是获得 R 个确认才算成功, 但是 Riak 并不关心这些确认是来自 primary vnode 还是 alternative vnode (这正是 “sloppy” 松散一词的由来)。如果 Riak 基于的是 *strict quorums* (严格仲裁) 则确认必须来自 primary vnode, 结果将使得系统在 primary vnode 下线或者不可达时可用性降低。

一旦开始在 replica vnode 间分发数据和状态, 就引入了不确定性。如何知道一个操作是否成功地复制到了所有节点呢? 如果由于分区、节点、网络、硬件或软件故障导致数据变得不一致会怎么样?

370 当节点未能协商一致, 返回了不同的值时, Riak Core 试图使用 DVV (Dotted Version Vectors) 机制来解决冲突。对于目标值的一系列写入事件, Riak Core 利用 DVV 能够部分性地判断出它们的顺序, 从而有助于判断哪一个值才是正确的。这种排序并非基于时间戳, 因为时间戳太不可靠, 并且难以跨集群节点保持同步, 它是基于在每个节点上的单调递增计数器的逻辑时钟的值。如果 DVV 信息还是不足以化解冲突, 则将该状态的所有冲突值都作为兄弟值返回给客户端, 交由客户端应用程序解决, 也许是使用特定于领域的知识来判别。

那么, Riak Core 是如何帮助我们实现分布式架构的呢? 虽然你的核心最多只能有 100 个节点, 但你可以将这些节点用作通向其他集群的集线器或网关, 如图 13-5 所示。运行 Riak Core 的逻辑节点创建了一个完全网状环, 用于消息传递、作业调度、路由请求到服

务节点，或作为其他集群的网关。

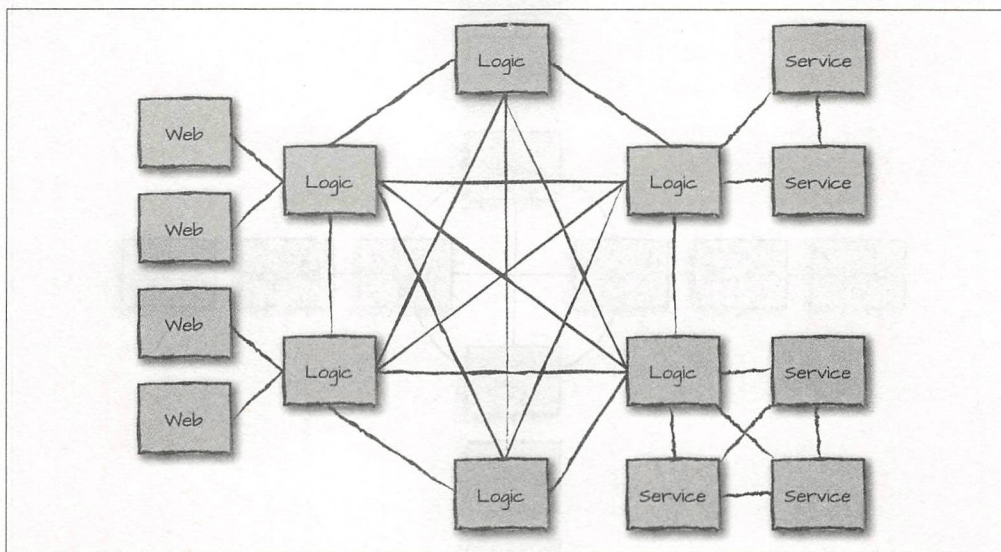


图 13-5: Riak Core 模式

图 13-6 使用另一种方法实现大规模可伸缩性：星形架构，其中彼此连接的服务节点可用于存储和分析，并基于负载动态增加和减小规模。这两种模式都有其目的，并克服了全网状网络遇到的可伸缩性问题。可以用更多更复杂的模式，同样也可以用简单的模式。比如运行多个 Riak Core 集群，它们通过隐藏节点作为网关相互连接。

如果一致性哈希和 Riak Core 适用于你正在解决的问题，那么你可能还想看看 **NkCLUSTER** 应用程序 (<https://github.com/NetComposer/nkclusterand>) ——其建立在 Riak Core 的基础上，用于创建和管理 Erlang 节点集群，并分发和安排集群上的作业。**NkDIST** (<https://github.com/NetComposer/nkcluster>) 是一个能够均匀分配进程的库，当 Riak Core 集群通过添加或删除节点进行重新平衡时，它能够自动移动进程进行平衡。你可以在 **NkDIST** 和 **NkCLUSTER** 二者的 GitHub 页面上分别找到文档。

371

如果想进一步阅读 Riak Core 方面的资料，我们推荐 Mariano Guerra 在 GitHub 上的 *Little Riak Core Book* (<https://marianoguerra.github.io/little-riak-core-book/>) 一书。可在 Basho 网站 (<http://docs.basho.com/>) 阅读 Riak Core 的官方文档 (Riak Core 正是由 Basho 公司创建和维护的)。在网络上搜索一番也能找到许多讨论和教程。最后，关于如何使用 Riak Core 的一个极佳的例子是 Udon (<https://github.com/mrallen1/udon>) ——一款由 Mark Allen 编写的分布式静态文件 Web 服务器。

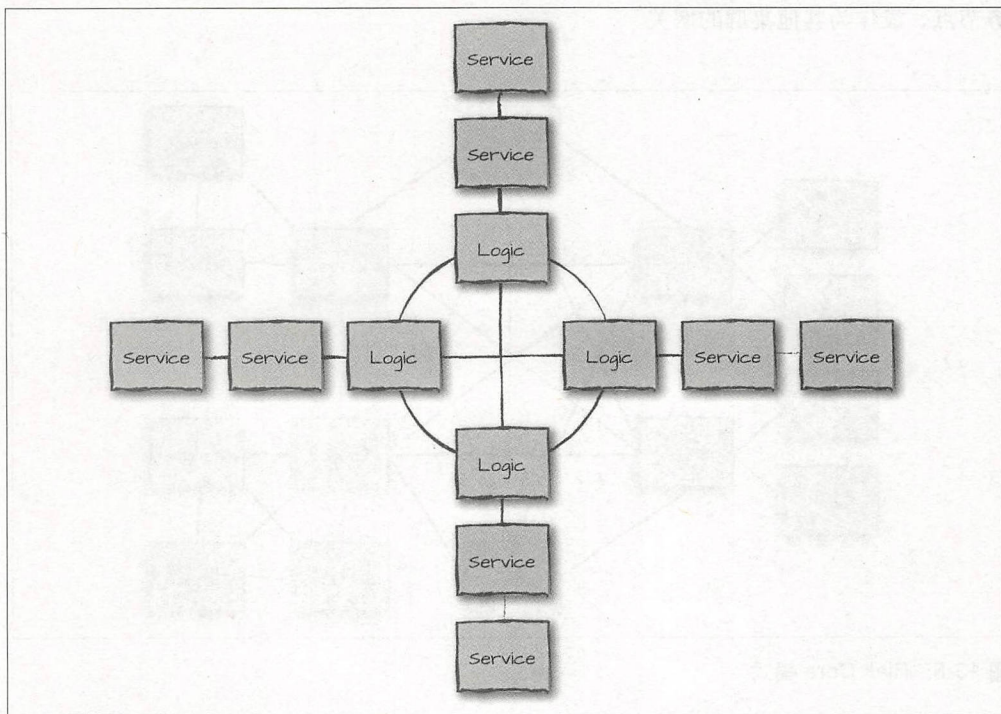


图 13-6: Riak Core星形

372 可伸缩的分布式 Erlang

可伸缩的分布式 Erlang (SD Erlang) 采用了与 Riak Core 不同的方法。SD Erlang 出自格拉斯哥大学的 RELEASE 研究项目。虽然在撰写本文时它尚未达到可用于生产环境的程度,但它背后的想法很有趣,并且已被证明允许系统扩展到数万个节点。基本的方法是通过现有的分布式 Erlang 做一个小的扩展来减少网络连接和命名空间。

SD Erlang 定义了一个称为 *s_group* 的新层。节点可以属于零个、一个或多个 *s_group*, 属于同一个 *s_group* 的节点会共享连接和命名空间。命名空间是使用分布式 Erlang 中的 `global:register_name/2` 函数或 SD Erlang 中的 `s_group:register_name/3` 函数注册的一组名称。在分布式 Erlang 中注册的名称会复制给所有相连的普通 (不隐藏) 节点。在 SD Erlang 中, 名称将在给定的 *s_group* 的所有节点上复制。

图 13-7 显示了名为 G1 和 G2 的 *s_group*。每个 *s_group* 包含三个 Erlang 节点。因为节点 C 由两个 *s_group* 共享, 所以它可以在不同的 *s_group* 中的节点之间传输消息。节点 C 被称为网关。

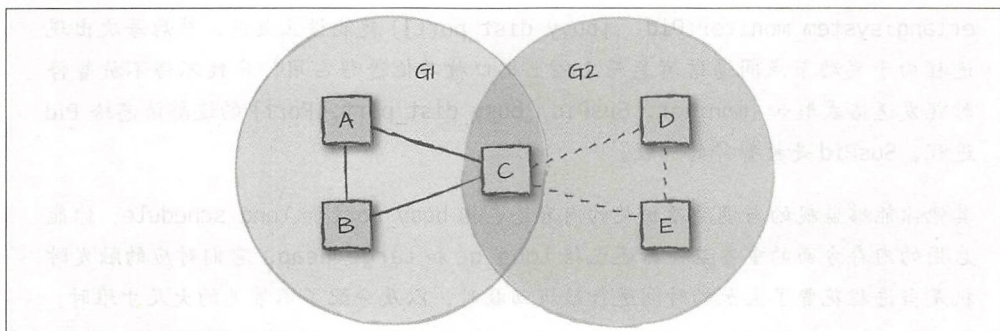


图 13-7: SD Erlang 组

使用 SD Erlang 的节点组概念，程序员可以按照不同的配置安排节点，例如指定节点所属集群，并通过网关连接它们。

为了使 SD Erlang 应用程序具有可移动性和可伸缩性，还引入了一种称为半显式安插 (*semi-explicit placement*) 的概念。它基于节点与周边节点的通信距离以及节点自身属性来控制节点的安插位置。节点属性包括节点的硬件特征、软件特征和程序员定义的特征等，使得它们能够了解到自身以及相邻节点的特点。通信距离是按照数据从一个节点传输到另一个节点所需的时间作为指标界定的。假设各个连接都具有相等的带宽，则较短的传输时间对应于节点之间较短的通信距离。

关于 SD Erlang 的文档可以在格拉斯哥大学的网站上 (<http://www.dcs.gla.ac.uk/research/sd-erlang/>) 找到，许多关于它的会议谈话和文章也可以在网找到。

套接字与 SSL

373

有时分布式 Erlang 还不够。对于超高容量的系统，瓶颈可以出现在全局名称服务器(global name server)、rex、网络内核——更不用说分布式的 Erlang 端口本身了，即使它速度很快，但毕竟一次也只能处理一个请求，因为它是设计用于控制消息的而不是传输数据的。或者，正如我们在 DMZ 示例中看到的那样，出于安全方面的考虑，你可能希望避免分布式 Erlang，限制全连接网络带来的过度开放性。当分布式 Erlang 不再适用于你的需求时，在 ssl 或 gen_tcp 库上添加微薄的一层也许是一个办法。你在节点之间打开一个或多个套接字，可控制发送和接收的信息流。

系统监视器

如何察觉分布式 Erlang 端口上出现了拥塞？在 Erlang/OTP 文档中有一个隐藏得很深的 BIF，允许你触发和内存管理与调度器相关的监视器事件。只需调用

`erlang:system_monitor(Pid, [busy_dist_port])` 就能设立监视。然后每次出现进程由于发动节点间通信消息而又碰上端口被其他进程占用，导致不得不被暂停时就发送格式形如 `{monitor, SusPid, busy_dist_port, Port}` 的追踪消息给 `Pid` 进程。`SusPid` 是被暂停的进程。

其他你能够监视的与调度器相关的内容还包括 `busy_port` 和 `long_schedule`。你能启用的内存方面的重要监视器还包括 `long_gc` 和 `large_heap`，它们对应的触发时机是当进程花费了太长的时间进行垃圾回收时，以及分配了不常见的大尺寸堆时。

对于在线系统环境，注意你处理系统消息的方式。我们已经见过不少糟糕的系统在重负载情况下每小时可以生成数百万条消息。你可以读一读 `erlang` 手册页中的 `system_monitor BIF` (http://erlang.org/doc/man/erlang.html#system_monitor-0)。我们还会在第 16 章进一步介绍系统监视方面的内容。

当使用套接字传输大量数据时也可能会出现瓶颈。例如，我们曾经参与过一个管理即时消息的系统。即时消息的特点是长度偏短，突发性强，所以单条来自 DMZ 的 TCP 连接便已足够应付极端负载。当我们升级这一系统增加了管理电子邮件功能时发现，当暴露于连续的重负载时，队列将迅速在前端节点中积攒起来。这与正在发送的消息的大小有关，这些消息远远大于即时消息，导致 TCP 套接字进程阻塞。阻塞最终导致虚拟机上运行的内存不足。而网络实际上远未饱和，我们在前端节点和逻辑节点之间增加多个连接（参见图 13-8）便摆脱了此瓶颈。

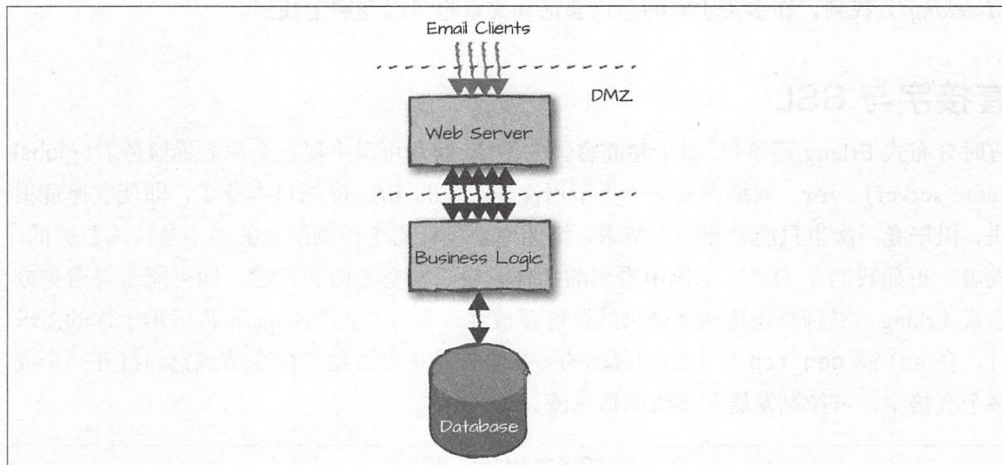


图 13-8：通信瓶颈

我们必须在节点间使用多个连接的典型场景，包括传输图像、日志、电子邮件和电子邮件附件等。但数据量必须足够多，多到使用多个连接确实能得到足够的回报，避免只是

过早的优化。从单一连接开始，只有当你的指标显示你遇到的问题可以借助多连接修复时，才那么做。

这是一种常见的方法，有几个开源库。GitHub 上的 `gen_rpc` (https://github.com/priestjim/gen_rpc) 应用程序已经进行过的基准测试显示它每秒能处理超过 60 000 个 RPC 请求。如果你需要简单的功能，还可以编写自己的连接 API。一种最简单的封装方案是，这样一个 API 只不过是一个薄层，由几十行对应用程序的流量和安全性要求进行高度优化的代码组成。也就是说，把你的套接字库构建在某种进程池库——比如 `Poolboy` (<https://github.com/devinus/poolboy>)——之上也许是有意义的。

通过图 13-2 所示的例子也可以看出，为什么从安全角度来说有时候不应当依赖于 Erlang 的分布式进程架构。我们可能不希望前端节点使用分布式 Erlang 与逻辑节点进行通信，因为一旦入侵者获得了对客户节点的访问权限，他就能获得对所有相互连接的节点的完全访问权限，并且能够远程执行操作系统级命令。想象一下某个热衷于执行 `rpc:multicall(nodes(), os, cmd, ["rm -rf *"])` 来获得内心平静与空空硬盘的家伙。

◀ 375

即使你在前端节点和逻辑节点之间采用了自创的基于 TCP 或 SSL 的通信库，仍然可以使用分布式 Erlang 来让逻辑节点相互通信，并通过 `Riak`、`Mnesia` 或简单消息传递来共享数据。反过来，逻辑节点可能会使用 RESTful 方法与服务节点进行通信。当你的系统开始变得复杂时，出于安全性、性能和可伸缩性考虑，采用混合通信方式并不稀奇。混合通信可以发生在节点、节点类型或节点家族之间。

面向服务和微服务的架构

创建可伸缩系统的另一种模式是微服务和面向服务的架构 (SOA)。虽然现如今 SOA 被一些人认为是重量级和老式的，但它被广泛应用于企业系统，其思想是微服务的基础。两者在概念上都类似于客户端 - 服务器范式，其中各个进程和节点（或节点家族）都会为其他节点和进程提供服务。这些服务（通常是独立的或松散耦合的）共同提供了你的系统所需的功能。通常我们习惯用术语“API”指代它们，这些服务（也可以叫函数 / 功能）能够响应其他节点的请求，执行相应的动作。这些服务所提供的功能其实与本书中你已经看到过的是是一样的。包括客户端前端接口、认证数据库、日志记录、警报、逻辑节点和其他服务节点（参见图 13-9）。服务应该以足够通用的方式被打包，不仅便于其他服务重用，而且能够跨系统重用。

众多服务通过服务总线 (service bus) 连接在一起。它们使用一种协议来描述服务如何交换和解释消息。这是通过服务元数据 (service metadata) 完成的，其描述了每个服务的作用及其所需的数据。这类元数据具有的格式应当使节点可以动态配置并发布自身的服务，进而也能动态发现并利用其他节点的服务。消息本身通常使用 JSON、XML、Protocol Buffers、Erlang 数据项，甚至 OMG IDL 进行定义。

◀ 376

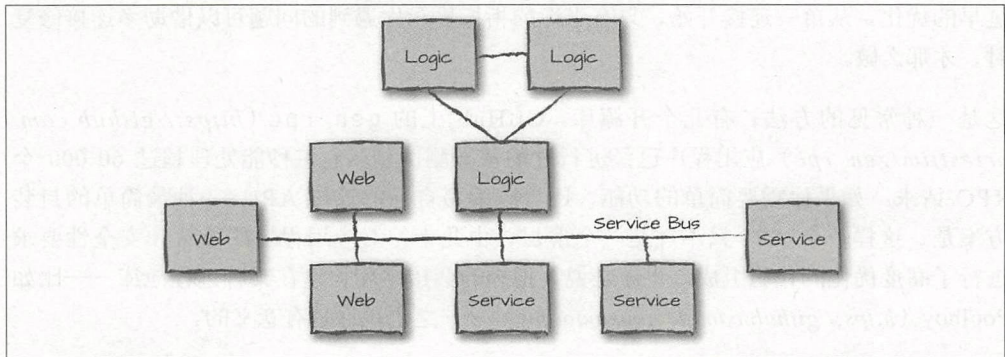


图 13-9: 面向服务的架构

服务总线运行于某一网络之上，并允许按照特定协议进行通信。可以使用 SOAP、HTTP 或 AMQP 来发送请求。你可以使用 Web 服务、Java RMI、Thrift 绑定，甚至是基于 Erlang 的 RPC 和消息传递。某些消息总线还具有一些额外的好处，比如限制请求量、调节负载和背压等。我们在第 15 章的“负载调节与背压”一节中将更详细地介绍这些概念。

标准化协议的优点是它们允许你把现成的组件或独立节点组合到一起，而这些组件或节点可能是以各种编程语言实现的。同时，它们强制你以便于跨系统重用的方式打包你的服务。然而，随之而来的代价是，跨节点共享的数据尺寸有所增加，以及请求和回复时额外的编码和解析开销也会增加。

gproc

gproc 是 Ulf Wiger 用于服务发现的 application。它提供了一个注册表，你可以在其中存储描述进程角色和特征的元数据。它允许你使用任意 Erlang 数据项来注册进程，并允许单个进程的多个别名。对于多项型（nonunique）进程属性，它允许你存储并借助匹配规范以及查询列表推导来查询。其注册表是全局性的，允许进程元数据的存储和访问分布在多个节点上。你可以在 GitHub 上找到 *gproc* 及其文档（<https://github.com/uwiger/gproc>）。

点对点

点对点（p2p）架构可能是所有分布式架构模式中伸缩性最强的一种，因为它是去中心化的，并且所有节点的类型完全相同，相互之间可按需建立临时性的连接。每个节点都具有相同的权限、功能和职责，这与客户端 - 服务器架构模式相反，其中某些节点类型以服务其他节点类型为目的。

而在 p2p 架构中，每个节点都既是客户端又是服务器，使得它能以去中心化的方式启动通信会话。想想 BitTorrent、Gnutella、Gossip 和 Kazaa 之类的协议。尽管对于很多人来说，p2p 就是文件共享的代名词，但在 Erlang 的世界中它更多地与大规模并行计算、分布式文件存储和大数据分析相关联。p2p 节点倾向于以较低开销、不可预测和快速变化的方式形成连接（参见图 13-10）。然而，通过多个节点传递数据才达到其最终目的可能会导致网络总体负载升高。

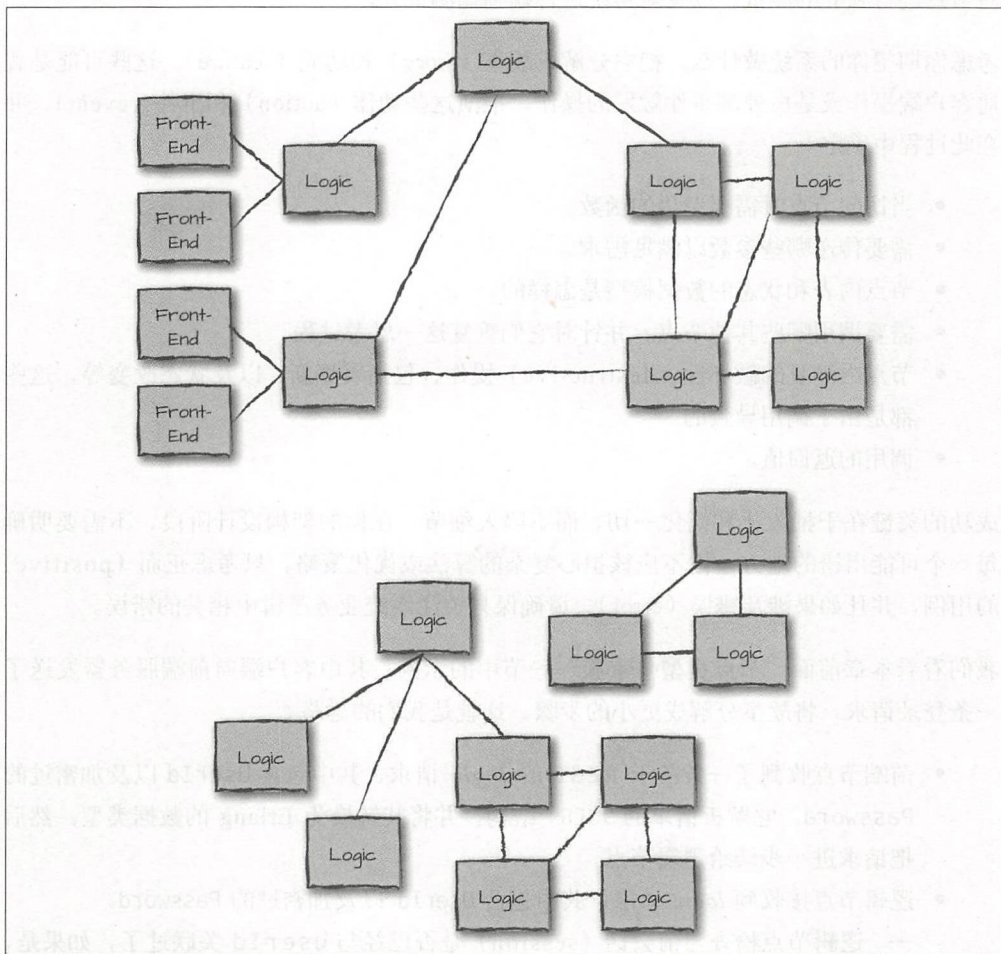


图 13-10: 点对点架构

话虽如此，没有任何理由可以阻止你使用 p2p 节点作为通信集线器——客户端可以用与 Riak Core 描述的架构模式相似的方式连接到它们。虽然你不会每天都和这些模式打交道，但是对于需要在分区网络中持续执行并且不需要强一致性的系统而言，这些模式还是不错的。

接口

一旦完成了将节点分解为节点类型，并定义了它们提供哪些服务，以及如何进行通信后，就可以指定节点导出的接口了。根据系统的大小和复杂性不同，这有可能会是令人生畏和沮丧的任务，特别是当你不知道从哪里开始、如何将其分解成较小的任务时。然而，这很重要，因为接口除了在发送请求时被其他节点使用外，它们也被用于实现业务逻辑、对节点进行独立的测试，以及对系统进行端到端测试等。

考虑你期望你的系统做什么，把它分解成故事（story）和功能（feature）。这些可能是普通客户端操作或是由外部事件触发的操作。推演这些动作（action）和事件（event），并在此过程中确定：

- 当访问节点时需要调用的函数。
- 需要传递哪些参数以满足需求。
- 节点内表和状态的数据模型是怎样的。
- 需要调用哪些其他节点，并针对它们重复这一思考过程。
- 节点内涉及的破坏性（destructive）操作，包括表更新，以及状态改变等，这些都是由于调用导致的。
- 调用的返回值。

成功的关键在于抽象化和简化一切，而不陷入细节。在你的架构设计阶段，不需要明确每一个可能出错的地方。你不应该担心复杂的算法或优化策略。只考虑正面（positive）的用例，并且如果涉及错误（error），请确保只关注系统业务逻辑中相关的错误。

我们看看本章前面“节点类型与家族”一节中的示例，其中客户端向前端服务器发送了一条登录请求。将故事分解成更小的步骤，这就是我们的思路：

- 前端节点收到了一条基于 REST 的 *login* 请求，其中包含 *UserId* 以及加密过的 *Password*。它解析请求的 JSON 结构，并将其转换为 Erlang 的数据类型。然后把请求进一步转给逻辑节点。
- 逻辑节点接收到 *login* 请求，其中包含 *UserId* 以及加密过的 *Password*。
 - 逻辑节点检查当前会话（*session*）是否已经与 *UserId* 关联过了，如果是，则对当前用户做重认证（*reauthenticate*）并返回已有的 *SessionId*。
 - 否则如果当前会话不存在，则逻辑节点将请求转给认证服务器执行认证，并返回 *SessionId*。
- 认证服务器收到 *auth* 请求，其中包含 *UserId* 以及加密过的 *Password*。
 - 如果认证成功、账户处于激活（*active*）状态，并且密码尚未过期，则服务器将确认该请求并返回与 *UserId* 对应的 *UserData*。

- 如果认证失败，认证服务器会返回失败原因 Reason。原因可能是 `unknown_user`（未知用户）、`bad_password`（密码错误）、`user_suspended`（用户被停用）或 `password_expired`（密码过期）等。
- 逻辑节点收到认证服务器返回的结果。
 - 如果认证成功，并且对应当前用户的会话尚未存在，则创建一个唯一的 `SessionId`，并把它与 `UserData`、`UserId`、`TimeStamp`（时间戳）一起记录在会话表（`session table`）中。最后给前端节点返回 `SessionId`。
 - 如果认证成功，但是当前用户已经有对应的会话存在，则直接返回已有的 `SessionId` 给前端节点。
 - 如果认证失败，逻辑节点返回 `login_failed`（登录失败）、`user_suspended`（用户被停用），或 `password_expired`（密码过期）等结果给前端节点。
- 前端节点收到逻辑节点返回的结果，创建对应的 JSON 结构然后回复给客户端。

我们将所有方面都保持在较高的层面上，只关心节点级别的函数调用和参数，并讨论可能在我们系统的业务逻辑中产生的返回值和错误。此刻请忘记解析错误、进程、节点崩溃和不可用，或是网络连接问题。但是请注意，如果登录失败，则逻辑节点只会概括性地描述失败理由，而不会具体指明到底是 `UserId` 还是 `Password` 不正确；这是一种安全措施，使攻击者更难确定是否存在特定的 `UserId`。

与接口定义一起，我们还会明确一些初始数据与状态。我们会在文档中注明，调用这些结构将会导致数据产生怎样的变化。完成这个练习后，表 13-1 列出了我们期望提取的内容。

表 13-1: 接口和表

Web 前端节点
<code>login(UserId, Password) -> {ok, SessionId} {error, login_failed}</code> 无表或状态
逻辑节点
<code>login(UserId, Password) -> {ok, SessionId} {error, login_failed user_suspended password_expired}</code> <code>SessionTable: SessionId, UserId, TimeStamp, UserData</code> <code>UserTable: UserId, SessionId</code>
认证服务器
<code>auth(UserId, Password) -> {ok, UserData} {error, unknown_user bad_password user_suspended password_expired}</code> <code>UserTable: UserId, Password, AccountState, TimeStamp, UserData</code>

◀ 380

针对所有的用例和故事这样做将为你提供坚实的基础，使得你能够以此为基础设计

各个节点，以及发现你可能错过了的其他故事和用例。如果许多用户参与了这个项目，他们有阅读高级（high-level）设计文档的需要，那么为这些函数完成的事情提供简要的介绍会对他们很有帮助。当你在反复设计系统接口的过程中，你会重新调整数据库表、将功能移来移去、尝试减少重复的数据等。不要以为你会在第一次尝试时就能成功。

总结

在本章中，我们介绍了确定你的系统的分布式架构的第一步。你必须在某些时候做出选择，并不断在实现和验证阶段重新审视这些选择。需要考虑的东西还有很多，所以要小心，避免迷失在细节中以及陷入过度设计的泥潭。如果你需要每秒处理 10 000 个携带小量数据的请求，则完全连接的分布式 Erlang 可能就已足够，但是如果你需要传输大量数据，则仅仅依靠分布式 Erlang 就不够了。不要陷入过早优化的陷阱，增加复杂性会降低系统运行速度，降低可靠性，并增加维护成本，而不会带来任何附加的好处。如果有疑惑，那么启动你的项目时进行小型的概念验证，以确保你的方法是正确的。这将能够验证你的想法，并防止你把错误带到生产系统中去。

本章我们是按如下步骤进行介绍的。

1. 将系统功能分割为一系列可管理的、独立的节点。

此过程有助于我们将节点分类为前端节点、逻辑节点和服务节点。尽量让每个节点提供简单的服务，并提醒自己节点是一种隔离故障的方式。丢失任何一个节点，都不应当影响到不经过该节点的请求。

- 381
2. 选择一种分布式架构模式。

在决定使用一种模式时，应当考虑到可伸缩性、可用性和可靠性。静态数量的节点是否就已足够，还是需要能动态伸缩？你真的需要一个分布式框架吗，还是一个简单的全连接分布式 Erlang 集群就已足够？虽然你需要从一开始就把可伸缩性和可用性设计到系统中，但不要过度设计。永远保持简单，只在必须时才增加复杂性。你知道如何使用 Riak Core 或者 SD Erlang 并不意味着你必须使用。问问自己，你需要解决的问题是否属于它们解决的范畴。

3. 为你的节点间、节点家族间、集群间通信选择合适的网络协议。

虽然大多数系统可以作为完全连接的分布式 Erlang 集群运行于防火墙之后，但也有一些情况你需要考虑。你是否需要优化网络、带宽、速度？你的安全要求是什么？最重要的是，你如何处理网络的不可靠性？对于运行于同一个机架内的若干节点，以及运行于地理位置遥远的若干节点所选用的方法应当是不同的。有些选择方案你

可能会较容易确定优劣，也有一些方案你需要通过适当的基准测试后才能确定。

4. 定义好节点的接口、状态、数据模型。

确定接口的过程，也是在检验当初你将系统拆分为可管理的独立节点时所做的选择。要设计出正确的接口和数据模型需要反复迭代，也就是对设计的反复调整。你会希望在节点间相互请求时，冗余数据尽量少，并且参数的数量和尺寸也尽量少。你会希望节点间 API 能够标准化，同时能够满足外部协议和接口。

接下来是什么

至此我们已经涵盖了节点类型、系统蓝图以及节点和节点家族连接，现在是时候看看故障的各种场景以及如何减轻它们带来的影响了。下一章将介绍由于软件、硬件或网络问题导致请求失败时的重试策略，以及这些重试策略与跨节点和节点家族的状态与数据的分区和分布方式的结合。

永不停止的系统

为了构造一个容错的系统，你至少需要两台计算机。随 Erlang 而来的分布式支持、无共享内存设计、异步消息传递功能等为你进行跨计算机复制数据提供了基础，因此即使一台计算机崩溃，也可让另一台计算机接管。好消息是，适用于单节点系统的错误处理、故障隔离和自我修复等技术在涉及多个节点时同样有效，进而你可以跨集群透明地分布进程，并与在单个节点时一样使用相同的故障检测技术。相较于其他语言通过编写库来弥补语义缺陷的惯常做法，Erlang 的这一方式使得创建容错系统更容易和更可预测。核心要点是，Erlang 本身并不会直接给你一套容错的系统——但是它的编程模型赋予了你这种能力，相比于其他技术你只需付出很少的努力即可。

在本章中，我们将继续解释 Erlang 系统中常用的分布式编程方法。我们将目光转向跨节点和计算机的数据复制和重试策略，以及构建永远不会停止的系统所需面对的妥协和权衡。不同的方案决定了不同的数据分布方式，以及影响到由于你的控制不当导致失败时，如何重试请求。

可用性

可用性的定义是系统在一段时间内的正常运行时间。高可用性指的是具有非常低的停机时间，软件维护和升级导致的停机时间也包括在内。虽然有些人声称达到了九个九的可用性^{注1}，但是这种说法通常不是建立在长期运行的基础上的。九个九的正常运行时间意味着每年只有 31.6 毫秒的停机时间！眨一次眼睛尚需这么短时间的 10 倍，更不要说出现一些什么小的错误了。Erlang/OTP 通常实际达到的正常运行时间是 99.999%，相当于每年停机时间（包括升级和维护）只有 5 分钟左右。

注1 英国电信发布了一篇新闻稿，声称其 AXD301 ATM 交换机网络在为期六个月的试用期间达到了九个九的可用性，该网络承载了其所有的长途通话。

高可用性是你的系统没有单点故障、能容错、有弹性和可靠的结果。高可用也可以是系统即使出现局部故障，仍然能够提供一定程度的服务——虽然水平有一定下降。我们来详细了解一下对你正在尝试构建的系统有用的术语的含义。

容错

容错指的是系统在出故障时能够以可预测的方式继续运行的能力。这种故障包括来自软件层面的错误，例如某些进程因为 bug 或状态无效而导致的崩溃。也包括来自网络或硬件层面的故障，或者是节点的崩溃。“能够以可预测的方式运行”指的是能自动寻找可应对请求的替代节点，或者只是将错误返回给调用者。

在图 14-1 所示的示例中，一个客户端向运行 Web 服务器的前端节点发送了一条请求。该请求被解析后转发给了逻辑节点（参见图 14-1 的第 1 部分）。这时，逻辑节点本身或是逻辑节点中的进程崩溃了（参见图 14-1 的第 2 部分）。如果我们运气不错，则前端节点会检测到此崩溃并收到错误信息。如果运气不佳，则只会在内部触发超时。当收到错误或超时，逻辑节点进而会将错误发回客户端。

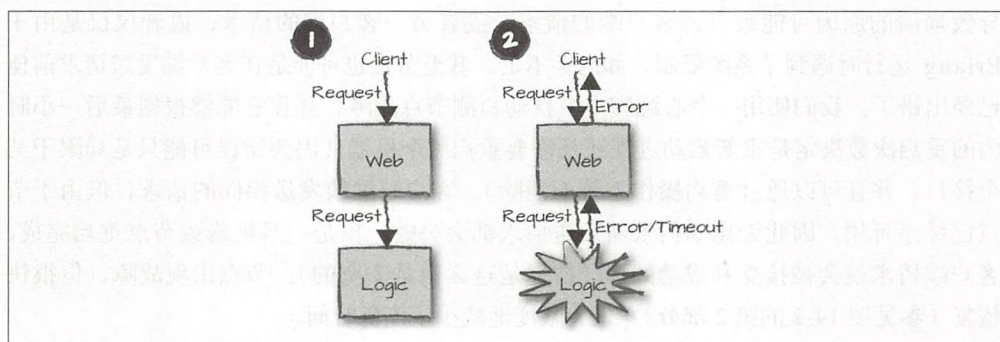


图 14-1：容错

此系统的运行方式是可预测的，而且我们还认为它是容错的，因为响应已经发送回了客户端。只要服务器、请求类型和协议允许，客户端可以以可预测的方式执行操作。客户端收到的响应可能不是它期待的那一个，但也算是有效的响应。现在轮到客户端决定下一步做什么了。它可能会尝试重新发送请求、传播失败信息，也可能什么也不做。

385

这个例子中最难的部分是搞清楚到底是不是逻辑节点故障，因为也可能是节点间的网络出现了故障——另外更糟糕的是，如果逻辑节点回应得过于缓慢，可能会触发前端节点的请求超时。慢节点和死节点之间没有实际的区别。你的前端节点需要了解所有这些情况，并处理这种不确定性。而这就需要借助唯一标识符、幂等和重试这三种基本手段，

我们将在本章后面讨论。审查日志和人为干预可能也是需要的。最后一件你希望的事是，当你的付款请求超时时，客户端能够持续地自动重试直到请求被确认。但这导致的可能后果是你一觉醒来发现同一本书买了 50 份。

Erlang 具有专用的、可跨节点的异步错误通道 (asynchronous error channels)。不管是节点崩溃还是进程崩溃，不管崩溃是在本地还是在远程节点上都没有关系。监视器、链接和退出信号，这些经过验证的错误处理技术，你既可以用在单个节点内，也可以用在分布式环境里。相较于来自本地而言，来自远程节点的退出信号的差别是存在延迟，而在异步消息传递机制中已经把这类问题考虑在内了。错误应当按照调用链传播，然后在每一个能够定位问题的地方做处理。这其中包括假阳性错误的处理——操作已经执行了，但在回报成功结果之前遇到了崩溃或超时。或者是由于网络问题导致超时，但是在收到超时提示之后，其实已经异步地操作成功了。这是异步分布式系统面临的最大挑战之一。

弹性

弹性是指系统从故障中快速恢复的能力。在图 14-2 所示的示例中，客户端向 Web 服务器节点发送了一条请求，但其在处理完请求之前就崩溃了（参见图 14-2 的第 1 部分）。导致崩溃的原因可能来自该客户端的请求，或者另一客户端的请求，或者仅仅是由于 Erlang 运行时遇到了系统限制，如内存不足。甚至节点也可能是在客户端发送请求前便已经出错了。我们使用一个心跳脚本来自动检测节点故障，并且它能够根据最后一小时内的重启次数决定是重新启动进程还是需要重启整个机器（因为错误可能只是局限于某个接口，并且可以通过重启操作系统而消除）。客户端继续发送相同的请求，但由于节点已经不可用，因此无论如何重新发送请求都会失败。但是一旦机器或节点重启完成，客户端请求就会被接受并成功处理（前提是这么做是安全的）。节点出现故障，但很快恢复（参见图 14-2 的第 2 部分），最大限度地减少了停机时间。

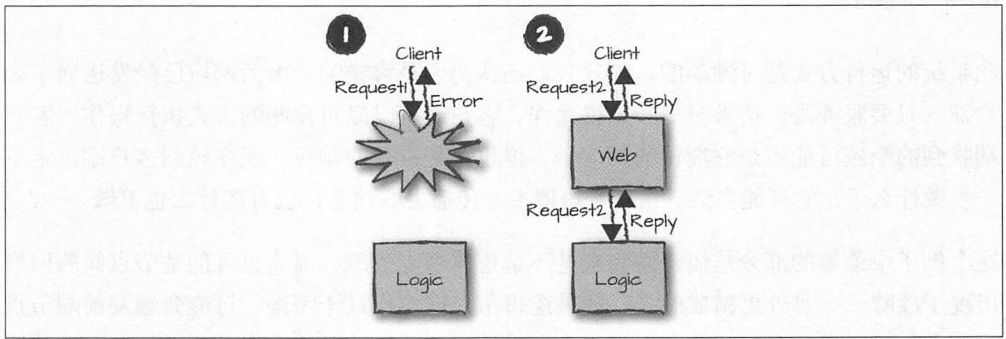


图14-2: 弹性

正如我们在本书前面的许多章节中看到的，关键在于隔离故障，将业务逻辑与错误处理分开。如果一个进程崩溃，依赖于它的进程也将被终止，然后该进程将重新启动。如果一个节点关闭了，心跳脚本会发现并立即触发重新启动。如果网络或硬件出现中断，则冗余网络将被使用。通过把功能划分成数量合理的不同类型的节点，使得隔离故障变成了一个简单、直接且容易的任务。如果某个节点集合了太多功能，这就会增加复杂性，从而增加节点崩溃的可能性，也就增加了恢复时间。

客户端内的退避算法

如果你的客户端程序具备碰到故障时自动重新连接并发送请求的功能，无论这种重试的频率如何，请确保在其中采用了退避 (*back-off*) 算法。想象连接着几百万个设备，每秒处理数十万次请求的系统出现了 1 分钟的中断，这将导致所有的设备都尝试重新连接并发送请求，结果造成了流量激增。只要你的系统没恢复，这样的浪潮将随着每一秒愈演愈烈。一旦你的系统恢复，如此巨大的流量又会将其击垮。如果处理不正确，这将导致更多的前端节点终止，产生更大的浪涌，然后继续摧毁剩下的部分。这就是我们所谓的级联故障 (*cascading failure*)，一种你必须综合考虑客户端和服务端才能防范的问题。

一种在客户端使用的最简单的退避算法变种是基于斐波那契 (*Fibonacci*) 数列的，其重试间隔从 1 秒增加到 2、3、5、8、13 秒，直至一个较大数字的上限，例如 89 秒、144 秒或者更多秒。而指数式退避算法 (*exponential back-off algorithm*) 是以指数方式增加请求间重试间隔；随机退避算法 (*random back-off algorithm*) 则适合于多个节点错开重试时。选择一种合适的算法将能够控制发生故障时大量重试伴随的流量激增，使得系统有机会恢复并继续运转。

可靠性

系统的可靠性指的是在特定预定状况下能够正常运行的能力。就软件和分布式系统而言，这些预定状况通常指的是失败和不一致等。换句话说，即使构成系统的组件出故障或数据因不能跨节点复制而变得不一致，系统也必须能够继续运行。当考虑可靠性时，你需要考虑这些组件的冗余问题。当我们提到组件时，讨论的不仅仅是硬件和软件，还包括数据和状态——它们需要在节点之间复制和保持一致。

单点故障意味着如果系统中的某个组件发生故障，则整个系统都失效了。该组件可以是一个进程、一个节点、一台计算机，或者甚至是将这一切连接起来的网络。所以为了使系统没有单点故障，不管是什么你都至少得有两份。至少有两台计算机运行着分布式软件，并且有相应的故障自动切换策略。你的数据和状态也至少有两份副本。两个路由器、

网关、接口,这样如果主设备出现故障,备用设备就能接手。同样的理由,备用电力供应(或备用电池)也得有。如果你很富有,请将这两台计算机分别放到单独的、地理位置相隔很远的数据中心。你还应该记住,任何东西都只有两份其实也会在某些情况下出现问题,因为如果其中之一出故障了,剩下的那个实例就又成了单点故障点。因此,当高可靠性是关键要求时,通常使用三个或更多个实例而不是两个实例。所有这些都会带来更高的带宽和延迟开销。

特殊手段

作者中的其中一位有一次刚抵达客户基站处,就看到一个挖掘机停在路边,旁边一位施工人员一脸茫然地抓住一根断裂线缆的两端试着把它们合在一起。最近一周以来,客户说互联网连接会突然丢失,甚至座机和移动电话也没了信号。原因是这一切用的都是屋顶上的同一根线缆。这个例子告诉我们,如果你希望服务请求在自然灾害(或者无知的人为错误)后还能继续可用,那么请确保基站资源一定要备有冗余。

美国监管机构的金融机构灾难恢复指南建议主要和次要数据中心之间的距离最小应为 200~300 英里。欧洲针对电信行业的建议则不那么极端,但是要保证如果核弹击中一个基站,或两个基站之间任何一处,至少要保证其中还有一个基站能够运转而不会带来影响!这就是为了获得高可用性而必须付出的代价。

最终,可用性变成了一个由成本、妥协、风险三项构成的问题。一个网络的中断所造成的经济损失可能低于安装一套备用网络/硬件所需的花费,这方面的思考实际上是一个商业决策问题。而本书是一本软件方面的技术书籍,所以让我们从精打细算中抽身,回到正轨上来。

对你的软件而言,一切都有两到三份意味着什么?请求抵达某个负载平衡器后,被转发到其中一个前端节点。选择哪个节点是由负载平衡器基于某种策略——随机、循环、哈希或是根据 CPU 负载、打开的 TCP 连接数量等——决定的。我们更喜欢哈希算法,因为速度快,并且具备可预测性和一致性,开销还比较低。当对请求进行故障排除(或追踪发生了什么)时,请求在节点间的路由关系如果是确定性的那么调试会容易很多,特别是当你有数百个节点和分散的日志记录时。

让我们来看一个如何避免单点故障的例子。前端节点接收到了请求,解析它,并将它转发给逻辑节点(参见图 14-3 的第 1 部分)。请求转发不久之后,出问题了。我们不知道故障到底发生在哪里,而且我们也无法确定请求本身的状态。我们不知道请求到底有没有到达逻辑节点,如果到达了,也不知道逻辑节点是刚刚开始处理它还是已经完成了处理。这个请求很可能导致了某个进程崩溃,导致了同步调用超时,甚至导致了整个节点

崩溃。或者也许节点没有崩溃，它可能是负载极重导致响应缓慢，或者只是与它相连的网络出故障了。我们应该能够区分节点崩溃和节点无响应两种情况。但除此之外，我们什么都不知道。

我们唯一能确定的是，有一个客户端正在等待回应。于是，在检测到错误后，前端节点将请求转发到次级逻辑节点。该节点处理请求（参见图 14-3 的第 2 部分），并将响应返回给前端节点，然后前端节点将其格式化并发送回客户端（参见图 14-3 的第 3 部分）。从始至终，客户端并不知道幕后竟经历了如此戏剧性的过程。正是发生错误的节点具备的弹性使它能恢复，重新连接到前端节点，并开始处理新的请求。因此，尽管系统出现了局部故障，但仰仗我们的“无单点故障”策略，依然为客户提供了 100% 的正常运行时间。

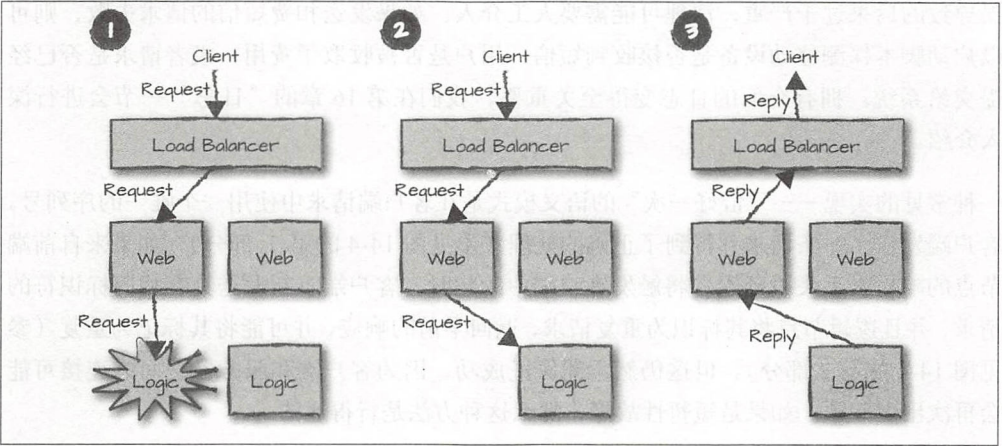


图14-3: 单点故障

最多一次、恰好一次、至少一次

389

当处理故障策略时，你需要用心确保所有特殊情况都被覆盖到。对于每个请求你有三种方法可以采用，因为你处理请求的方式对应着分布式系统中的节点之间消息传递的语义。在图 14-3 所示的例子中，你唯一能被保证的就是请求至少被执行了一次。如果你登录系统，并且第一个逻辑节点太慢导致前端节点尝试另一个逻辑节点并成功，那么最坏的情况是导致你实际登录了两次，并创建了两个会话，其中一个最终会超期。

同样，如果你发送短信或即时消息，你可能会对“最多一次”方式感到满意。如果你的系统每天发送数十亿条消息，则相对于负载和与为了做到可靠投递所需的巨大成本，丢失几条消息是可以接受的。你发送请求，然后撒手不管。在我们的无单点故障的例子里，前端节点把请求发送给逻辑节点后立即向客户端发送了回复。

但是，如果你要发送的是钱（即转账）或扣费短信怎么办？丢失资金，多次转账，或由于错误多次发送和收取相同的扣费短信，会让你陷入麻烦。在这种情况下，你需要用“恰好一次”的方式。请求可以成功或失败。如果错误出现在你的业务逻辑中，例如把优惠短信错发给了用户，这种情况下我们实际上认为请求是被正确处理的，因为它的返回值是正确的。应该担心的错误是超时、软件错误或状态错乱，导致进程或节点异常终止，从而使系统处于潜在未知或未定义的状态。只要在单个节点中使用的是“恰好一次”的方式，就可以检测到异常的进程终止。一旦你走向分布式，请求的语义就无法被保证了。

成功的情况是发送请求并收到响应。但是如果你没有收到响应，是因为请求从来没有到达远程节点、远程节点中出现错误，还是因为成功执行后的确认和回复在传输中丢失？于是系统可能处于不一致的状态，需要清理。在某些系统中，清理将由脚本自动执行，该脚本会尝试确定出现了哪些问题并定位它们。在其他情况下，比如代码太复杂或是错误导致的后果过于严重，清理可能需要人工介入。如果发送扣费短信的请求失败，则可以启动脚本探测移动设备是否接收到短信、用户是否被收取了费用，或者请求是否已经提交给系统。拥有全面的日志变得至关重要，我们在第 16 章的“日志”一节会进行深入介绍。

一种常见的实现——“恰好一次”的语义模式是在客户端请求中使用一个唯一的序列号。客户端发送了一条请求并得到了正确的处理（参见图 14-4 的第 1 部分）。如果来自前端节点的响应被丢失或延迟，将触发客户端中的超时。客户端重新发送具有相同标识符的请求，并且逻辑节点将其标识为重复请求，返回早前的响应，并可能将其标记为重复（参见图 14-4 的第 2 部分）。但这仍然不能保证成功，因为客户端和服务器之间的连接可能会再次出现问题。如果是短暂性故障，那么这种方法是行得通的。

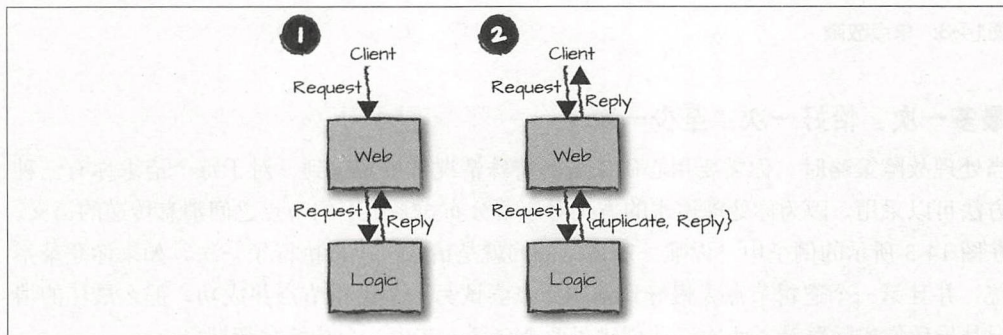


图 14-4：重复请求

这种方法依赖于幂等性。这个术语描述了用户可以重复多次施加同一操作而结果与只施加一次等价。例如，对于更改客户收货地址的请求，系统成功执行一次或多次结果都一样，当然这里假设每次请求的收货地址是相同的。这样的请求实际上可以执行多遍，因

为第二次以及后续执行基本上都没有明显的副作用。当然，由于我们采用了请求标记方案，其实第二次和后续的执行根本不会发生。 < 391

想象一个发送扣费短信的计费系统。你需要保证，如果你对用户收费，只能收一次，并且是特定用户收到短信后才收。通常用于保证此结果的方法是在发送短信之前在收件人账户中预留出部分资金。计费系统预留出这部分资金的同时，返回一个唯一的标识符。然后短信被发出，当然可能发送多次。仅当计费系统第一次接收到短信已送达的通知时才进行收费。执行收费时要使用那个唯一的标识符。后续如果尝试使用相同的标识符进行收费——例如一种可能的原因是同样的短信被复制投递了多次——也不会导致重复扣费。并且如果短信根本没有抵达接收者，先前预留的资金会在超时后自动释放。超时的发生也使得标识符不再有效。

最多一次、至少一次、恰好一次，三种方式各有优缺点。在决定使用哪种策略时，一定要记住，请求本身以及支撑请求处理的消息传递基础设施是不可靠的。这种不可靠性需要在每个请求的语义和业务逻辑中进行管理。最容易实现，也是最节省内存和 CPU 的方式是“最多一次”，你只需发送请求然后就什么也不用管了。出了差错使得你丢失了该请求，既不会影响性能，也不会影响到其他已成功请求。“至少一次”方法更昂贵，因为你需要存储请求的状态，监视它，并在收到超时或错误时将其转发到另一个节点。伴随着更高的内存消耗和 CPU 使用率，它还可能产生额外的网络流量。理论专家会认为，“至少一次”的方法不能保证是成功的，因为接收到请求的所有节点都可以关闭。我们会让他们刮目相看，让他们明白什么叫双重甚至三重冗余。最难策略是“恰好一次”，因为在执行有副作用的操作时需要提供保证。请求可以成功或失败，但绝不会在二者之间。

这样的保证对分布式系统而言是不可能的，因为可以出现请求被成功执行，但其确认和回复丢失的故障。你需要使用一些能够通过日志回溯调用过程的算法，了解发生故障的位置，以便尝试更正或补救。在一些系统中，这一过程非常复杂，或者风险很大，因此需要人工干预。

到目前为止，我们都在说，“任它崩溃”。是的，任它崩溃，而且无论你选择三种策略中的哪一种，都要努力进行恢复，确保在失败后，你的系统能恢复到一致的状态。Erlang 的错误处理和恢复的优雅之处在于，不管面对的是哪个方面的故障恢复——错误、软件、硬件、网络——你的恢复策略都是一样的。如果你的方法正确，那么不需要为了能够在崩溃、网络分区、丢包后重建状态而在进程中重复编写类似的代码。 < 392

数据共享

当你在考虑避免单点故障和故障恢复的策略时，还必须考虑在节点、节点家族和集群之

间复制数据的策略（如果需要这么做的话）。你的决定将影响你的系统的可用性（包括容错能力、弹性和可靠性），以及最终影响到可伸缩性。幸运的是，你可以将这些决定中的一部分推迟到对系统进行压力测试和基准测试时。你可能希望根据你已经了解的需求和过去设计类似系统的经验，做出其他决定。但无论怎么决定，访问和移动数据总是分布式系统中固有的最难处理的事情之一，如果处理不当，将有可能成为严重的瓶颈。例如以表 and 状态来说，你有三种可选择的方案：无共享（share nothing）、部分共享（share something）和完全共享（share everything）。应根据目标规模和可用性级别选择最为匹配的数据复制策略。

无共享

无共享架构中没有数据或状态被共享。针对的可以是节点、节点家族或集群。只要具备合适的基础设施（如硬件、网络和负载均衡等），无共享架构可以打造出线性可伸缩的系统。因为各组节点都有一份独立的属于自己的数据和状态，因此能独立运作。当需要伸缩时，你只需添加更多的基础设施并重新配置负载均衡器。

图 14-5 中的第 1 部分展示了两个前端节点和两个逻辑节点。*Client1* 和 *Client2* 通过发送登录请求提交凭证（credentials）创建了会话。请求被转发到两个前端节点中的某一个，具体转发给哪一个与负载均衡器中配置的策略有关。在我们的例子中，两个前端节点各拿到一个请求，然后转发到主逻辑节点。逻辑节点检查每个客户端的凭据并创建会话，将会话状态存储在数据库中。

之后某时刻，*Client1* 恰好在存储其会话的节点崩溃（相关会话数据全部丢失）之后发送了一个新的请求（参见图 14-5 的第 2 部分）。于是前端节点将其转发到备用逻辑节点，最终该逻辑节点拒绝了该请求，因为它找不到与该请求相对应的会话记录。客户端在收到“未知的会话”错误后，发送了新的登录请求，该请求转发给第二个逻辑节点处理。来自该客户端的所有后续请求如今都应该被转发到包含对应会话数据的逻辑节点。如果不这样做，等到先前崩溃的逻辑节点再次上线，客户端将不得不再次登录（参见图 14-5 的第 3 部分），此处我们简单地假设备用逻辑节点中的会话最终会自己超时并被删除。

由于我们不必在节点之间复制会话状态，因此可以获得更好的可伸缩性。随着并发连接用户数的增加，我们可以持续添加前端节点和逻辑节点来应对。这个策略的缺点是，如果你丢失一个节点，就丢失了其上的所有数据。在我们的示例中，一个节点上所有会话的数据丢失后，导致用户不得不再次登录，在另一个节点中建立新的会话。你还需要选择如何在节点之间路由请求，确保每个请求都路由给存储了相应会话数据的逻辑节点。这样才能保证节点出故障并恢复后的连续性。

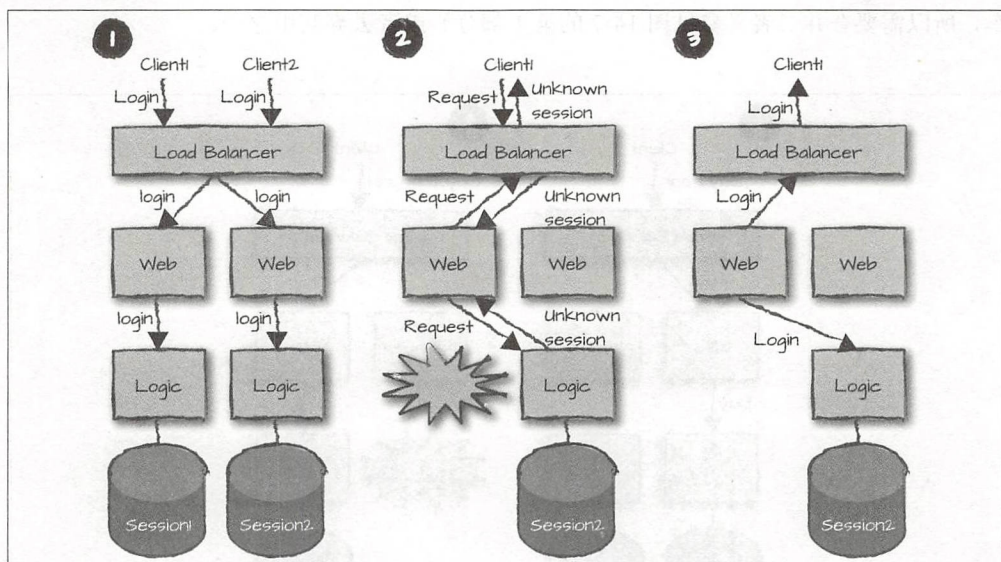


图14-5: 无共享 (share-nothing) 架构

部分共享

如果我们能做到节点出故障后用户仍然能保持登录，即会话有效，该怎么办？在“部分共享”架构中你可以复制一部分但并非全部的数据来解决问题。在图 14-6 中，我们跨所有逻辑节点复制会话状态。如果节点终止、缓慢或无法到达，请求将被转发给具有会话数据副本的逻辑节点。这种方法保证了客户端在切换逻辑节点时不需要再把登录过程走一遍。但是它会降低一些可伸缩性，因为会话数据需要在每次客户端登录时复制到很多节点，并在会话终止时从很多节点删除。每当在集群里添加新节点或重启节点时，开销会变得更大，因为其他节点上的会话数据可能都得复制给它并保持一致。

上面描述的策略被称为“部分共享”的原因就在于它是一种妥协：你复制的只是与各个会话相关联的数据和状态中的一部分，而非全部。该策略降低了复制开销，提高了容错能力。让我们回到先前电子商务网站的例子。会话数据被复制了，因此如果节点丢失，用户不必再次登录。但是，购物车的内容没被复制，因此在丢失节点时，用户意外地发现购物车被清空了。当用户正在结账并支付选购的商品时，只有当前可用逻辑节点中的那部分商品还在。

我们一直在假设的是，如果逻辑节点崩溃则它的所有数据都将丢失。可如果购物车是被存储到持久性键值存储中，一旦重启就又能读取了呢？再如果发生的是网络中断，或节点只是响应缓慢之类的，然后被我们误认为是死亡呢？当节点再次可用时，你需要考虑好路由策略——即新的请求要发给两个逻辑节点中的哪一个。并且由于你有两个购物

车，所以需要合并二者（参见图 14-7 的第 1 部分）或者丢弃其中之一。

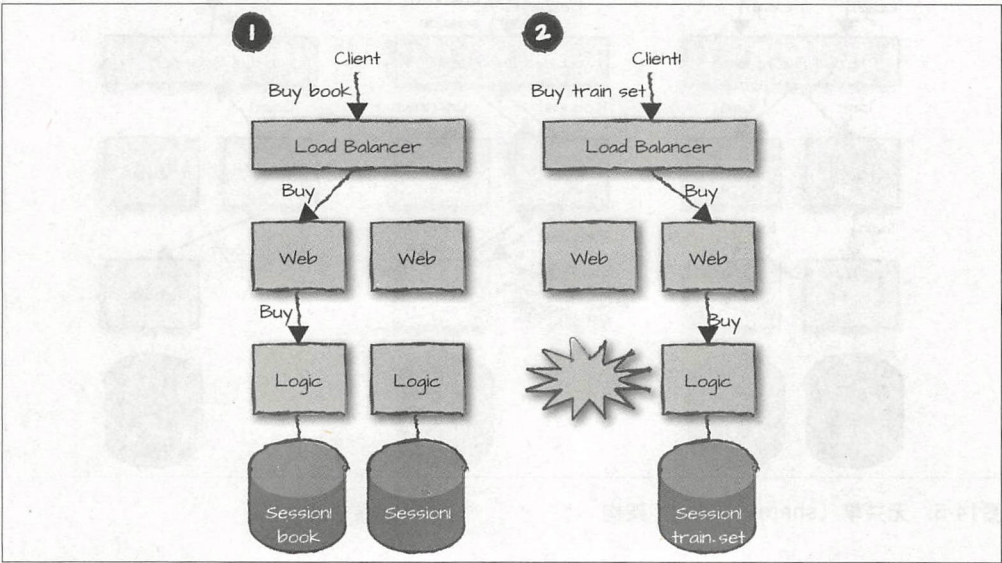


图14-6: 部分共享 (share-something) 架构

怎么做？把所有商品合并到一起？万一我们在第二个节点的购物车里添加了商品怎么办？会不会最终导致每种商品都买了双份？或者我们向第二个节点发送了一个删除商品操作，但因为该商品不在那里于是操作失败？如何解决这些问题取决于你的业务特点和风险情况。一些分布式数据库，如 Riak 和 Cassandra，在这方面为你提供了选项。在我们的示例中，崩溃的节点恢复后再次成为主节点，我们将第二个购物车中的内容移了过去（参见图 14-7 的第 2 部分）。

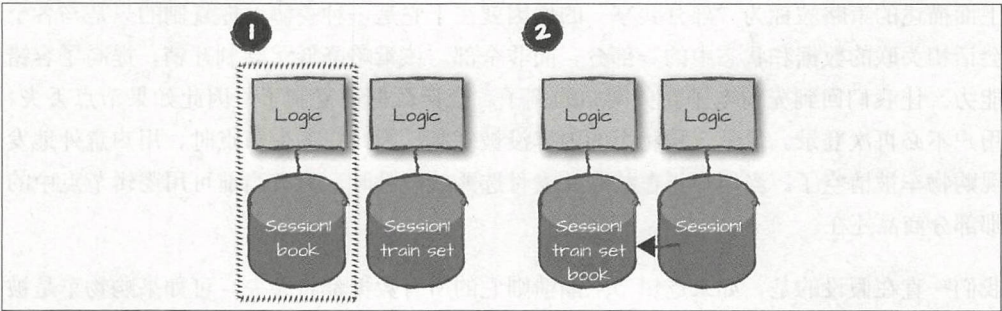


图14-7: 网络分区 (network partition) 与部分共享 (share-something) 架构

我们在第 13 章的“Riak Core”一节提到的 Dynamo 的论文中介绍了 Amazon 是怎么处

理故障恢复后的购物车合并问题的。如果在合并期间对商品是否该删除无法确定，则将其保留，将责任留给查看最终订单时的购物者，他们可以将其移除，也可以在发货后申请退款。有多少次，在结算的时候，你发现你的亚马逊购物车里又冒出了原本你已经移除了的商品？我们碰到过几次。

部分共享架构比较理想的场景是允许偶然丢失少数请求但是需要为某些更加昂贵的操作维持状态的时候。我们用了购物车的例子。换成即时消息服务器再考虑一下。最昂贵的操作，同时也是最大的瓶颈，是用户登录并开始会话。会话中需要访问身份验证服务器，检索用户的联系人列表，并向所有人发送状态更新。想象一个处理百万用户的服务器。当网络分区或节点崩溃后这 100 万用户重新登录，系统要恢复将需要发送 3000 万个离线状态更新（假设每个用户有 60 个联系人，一半在线）。

一个很好的解决方案是跨多个节点分布会话记录。然而，不共享状态通知和消息。你允许它们通过单个节点，以最多一次的方式发送消息，这样可以维持速度。假设节点崩溃，或者由于网络错误导致其与集群的其他部分分离，这时对于通知和消息，要么延迟传递，要么部分或全部丢失。你碰到过多少次发短信给邻近的人，结果他们在几个小时（或几天）之后才收到，甚至永远收不到的情况？

一致性

在处理分布式系统时，根据可见性（visibility）、有序性（ordering）、副本协调性（replica coordination）的不同，一致性可以细分为多种不同的形式。在一个完美的系统中，所有的节点都能够在相同的逻辑时间内以相同的顺序看到全部更新；不会出现读取到陈旧数据的情况；并且不会有异常的延迟、崩溃的节点、网络分区、消息丢失等。但在我们并不完美的真实世界中，这些问题会发生，因此我们的系统必须在一致性、可用性和延迟之间做出取舍。

一种较弱的一致性称为最终一致性（eventual consistency），在这种状况下，允许各个副本上的更新顺序不同，并且允许读取时返回陈旧的数据值。尽管这种模型听起来弊大于利，但实践中对那些要求读写可用性高以及延迟可预测的系统来说，这种模型很有价值，因为基于这一模型的系统即使在部分故障的情况下也可以通过读取陈旧数据的方式继续运转。

其他形式的一致性，比如单调读（monotonic read）和单调写（monotonic write）模型，则是与崭新（recency）程度有关。当你使用单调读模型读取一个值时，可以确保永远不会再见到比此次读取的更老的值。同样的，使用单调写模型，可以确保此刻发起的更新操作，一定先于后续发起的（针对相同值的）更新操作完成。这些有序性的代价是，分布式系统内需要更复杂的协商机制，从而潜在地增加了延

迟并降低了可用性。

读自写一致性 (read your own writes consistency) 则提供了更强的有序性保证, 就像其字面意思所表达的那样, 其组合了单调读与单调写, 对于一个值执行了更新操作后, 你就永远不会读取到这个值的先前版本。

通过使用 Paxos、Zookeeper Atomic Broadcast (ZAB)、Raft 之类的共识协议 (consensus protocols) 甚至能达到更高程度的一致性。在这类协议里, 更改某个值时, 必须获得大部分副本的投票同意。这些协议可以提供更强大的一致性保证, 但是为了实现这一点需要在副本间做更多的协商, 因而可能给延迟和可用性带来负面影响。即便如此, 如果你的应用确实需要这种程度的一致性保证, 使用一种已经被证明过的共识协议也要远好于自己发明。例如, Riak Ensemble (https://github.com/basho/riak_ensemble) 实现了 Multi-Paxos 协议——一种基本 Paxos 协议的改进版。

有时人们会困惑上述分布式系统一致性级别与我们在事务性数据库中常说的 ACID (Atomicity, 原子性; Consistency, 一致性; Isolation, 隔离性; Durability, 持久性) 概念中的 C 的区别, 二者是不同的。在 ACID 中, 一致性的含义是事务的效果必须在事务完成后才可见, 并且不会违反任何数据库约束。

397 完全共享

“不共享”和“部分共享”架构可能适用于某些系统和数据集, 但如果你希望使系统尽可能容错和有弹性, 该怎么办? 虽然现实中不可能拥有绝对不丢请求的系统, 但你确实可能会需要面对类似处理金钱、股票或其他操作的系统, 其中丢失任何一条请求导致的风险都是不可接受的。每项事务必须准确地只执行一次, 其数据必须强一致, 并且其操作必须要么完全成功要么完全失败。虽然你可以不在意一两条短信或即时消息的漏收, 但对于已经执行了的股票交易却消失无踪, 或者账户里的钱莫名丢失这类严重的状况, 这些模型无法为你防御这些状况。在这样的场景下, “完全共享”进入了我们的视野。全部的数据在所有节点之间共享, 任何一个节点都可以在硬件或软件出故障时接管请求。如果对请求的结果有任何不确定性, 则向用户返回错误信息。当事情出错时, 事后必须进行协调。例如, 如果你尝试从多个自动取款机中超额提取出你账户中的资金, 那么虽然你将获得这笔钱, 但之后银行将会对你的账户做出惩罚。但是, 通过使用冗余硬件和软件消除单点故障, 此类错误的风险可以被降至最低。

在图 14-8 中, 我们在两个逻辑节点中维持重复的会话和购物车数据, 每个节点只应对部分客户端, 并将会话状态和购物车数据复制给其他节点。如果一个节点终止了, 另一个节点就会接管。如果故障节点恢复, 则直到活跃节点的所有数据已被复制给刚恢复的节点并相互一致为止, 系统才会接受其他请求。



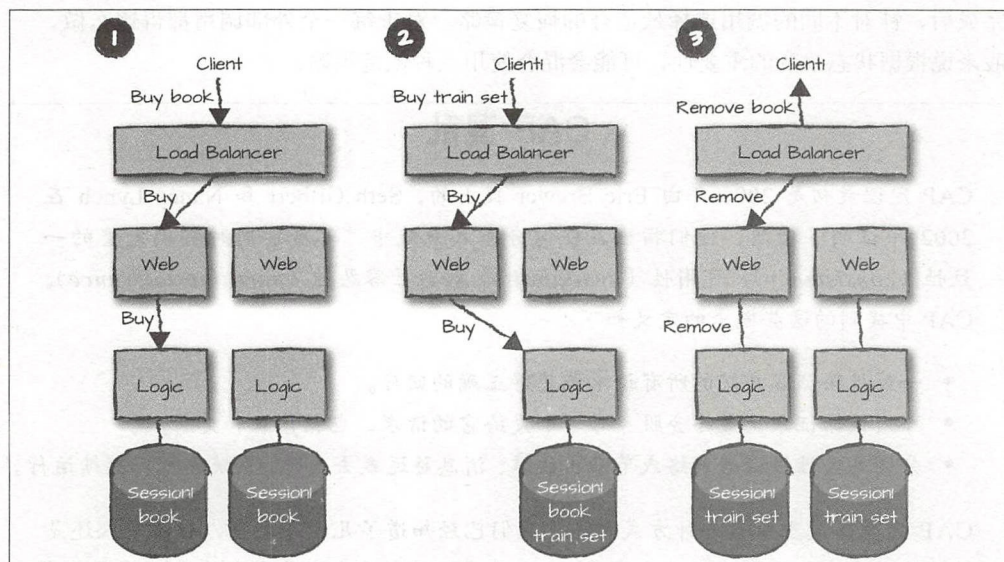


图14-8: 完全共享 (share-everything) 架构

我们称之为**主-主复制模式**。这与**主-从复制模式**不同，因为主-从复制中只有一个主节点负责数据。从节点可以读取数据，但是如果修改数据，例如插入或者删除之类，则必须要由主节点负责协调。如果主节点丢失，则系统将完全停止工作，或者在不允许进行写入和更新的情况下提供降级服务，或者选择其中一个从节点接手为主节点。

398

“完全共享”架构是所有数据共享策略中最可靠的，但这种可靠性是以牺牲可伸缩性为代价的。它能容忍节点的丢失而不会影响数据的一致性，但是如果某些节点出错，它也会丢失可用性。这种策略也是运行和维护成本最高的，因为每个操作都会产生额外的计算以及网络上的多次请求，为的是确保数据被复制并保持一致。每次重启后，从节点必须连接到主节点并检索数据以使自身维持与主节点的同步，确保它们具有正确的和最新的状态和数据。

虽然“完全共享”架构并不一定要求跨节点的分布式事务技术，但当需要处理诸如钱或股票等你承担不起丢失后果的数据时，你会需要这类技术。这与我们看过的消息收发时的需求形成了对比，其中基于最终一致性通过复制消息能够极大降低节点出故障时消息丢失的风险，但是它并不能保证你永远不会丢失消息。

当决定采用哪种数据共享策略后，你需要推演 API 中的各种请求，跟踪可能的调用流程，尝试找出可能出问题的地方。在节点内，针对同步调用，要考虑出现超时和非正常进程终止会有怎样的行为。对于异步消息，请确保对消息丢失（当接收的进程已经终止时）有正确的处理。跨节点，你需要考虑网络错误、网络分裂、慢节点和节点终止。这些都



完成后，针对不同的调用选择最适合的恢复策略。对于每一个外部调用都得这么做，一般来说根据状态改变的重要性，可能会混合使用三种恢复策略。

CAP 混乱

CAP 定理最初是 2000 年由 Eric Brewer 提出的，Seth Gilbert 和 Nancy Lynch 在 2002 年证明了猜想，他们指出在任何分布式系统中，不可能同时提供完整的一致性（consistency）、可用性（availability）和分区容忍性（partition tolerance）。CAP 中提到的这些概念的定义如下：

- 一致性保证客户端的所有请求都获得正确的回应。
- 可用性保证系统最终会服务每一个发给它的请求，包括读取和更新。
- 分区容忍性保证当网络或节点出故障、消息延迟或丢失时，系统也可以继续运行。

CAP 定理本质上以另一种方式讲述了我们已经知道了几十年的事，但许多人还是认为它是有争议和混乱的。这源于 CAP 经常被解释为要求你在设计分布式系统时从三个属性中选择两个。因为这三个属性中的一个——分区容忍性，属于分布式系统本身固有的，因而也就自动为你选择了，于是剩下还能选的就只有一致性和可用性了。举一个例子，有人声称 Mnesia 属于 CA 类型的系统，但很显然他们从未体验过 Mnesia 在网络分区时的表现（继续运行，依然可用）。

真正的分布式系统权衡绝非只是 CAP 难题展示的“三选二”那么简单。1977 年，CAP 出现之前几十年，Leslie Lamport 为分析系统属性引入了 safety 和 liveness 这两个概念，然后 Lamport、Bowen Alpern、Fred B. Schneider 等人在接下来的数十年间进行了深入的探索和解释。简而言之，safety 意味着，在分布式系统运行过程中，没有什么坏事发生；而 liveness 则意味着最终会有好事发生。CAP 中的一致性属性属于 safety 方面，因为它意味着正确性（correctness），而可用性则属于 liveness 方面，因为它意味着客户端永远会获得有效的回复。

20 世纪 80 年代，我们还得到了 Fischer-Lynch-Paterson (FLP) 不可能的结果，这证明了在一个异步环境中，即使系统只有一个部分故障的情况下也不存在一种分布式算法能够达成共识（consensus）。这一定律以及 CAP 中的 P 都表明，延迟与故障是分布式计算系统天生的特点，无论是硬件或是软件层面都是如此，因而永远不能小看或是忽视它们。面对失败时，由于某些节点无法抵达而无法达成共识，这反过来又意味着整个系统的统一性（agreement）、一致性和有效性会遭受一定程度的降级。无论你怎么分析它，都会遇到这些根本的事实：实现完全的 safety 和 liveness——或者换用 CAP 术语来说，即一致性和可用性——是在任何实际的分布式系统中都不可能实现的。



在现实生活中的系统中，如何在一致性和可用性之间做取舍实际上不仅与你的应用具体是什么紧密相关，而且即使在同一应用内的不同部分，也常常需要做出不同的取舍。举一个例子，在本书写作时流行一种健身追踪器。这种设备，用户佩戴后，能够收集与健康相关的数据，比如脉搏频率、健身活动的持续时间，或者跑步和行走时的步数，并把它们发送给设备厂商。厂商不仅把数据通过 Web 和移动端 App 展示给用户，还会展示到用户的社交网络甚至是特定的医疗保健商。尽管所有的数据可以存储在单个数据库中——因为处理用户注册的整个程序需要强一致性，要确保两个用户不会注册为相同的用户名——但是考虑到数据传输部分，拥有一个高可用的数据存储要比为感兴趣的各方提供完全一致的更新更重要。

这些应用解释了为什么一些数据库能够让应用按需选择，例如 Riak 能够同时支持强一致性和最终一致性。并且现代研究——例如 Peter Bailis 所做的工作深入分析了 consistency-availability spectrum（一致性-可用性频谱）——表明了应用程序经常能够以比此前认为的更少的一致性和协调性正常运行，在某些情况下甚至可以正确完成一些本来以为必须具备完全分布式支持才能做到的工作。

CAP、safety 和 liveness 以及其他相关的方法都是对分布式系统涉及的各方面权衡和选择的解释。由于他们的电话领域（telephony）研究背景，Erlang/OTP 的设计者们意识到了这些选择，但是随着 Web 领域的发展，大量网站的兴起迫使行业中的许多人都需要掌握这些分布式系统的内容，因为一旦涉及伸缩，这些问题统统都会出现，无论你喜欢与否，而且常常让人抓狂。

一致性和可用性之间的权衡

曾经我们正在重构一个系统，客户说他们从未断电，以 100% 可用性服务了所有的请求，包括软件升级时也是，而且多年如此。他们没有使用 Erlang，而且为了精益求精，把所有的一切都直接运行在大型机上！当我们开始揭开表面时，发现他们对可用性的定义只不过是前端节点始终处于运行状态，能够接收和确认请求而已。当逻辑节点和服务节点出现错误和中断的情况下，请求会被记录下来然后人工进行处理！我们可以认为，这个系统确实是高度可用的，但是它并不可靠，因为它无法永远按照预定的方式运行。它在出现故障后必须人工干预才能回到一致状态。你在恢复策略中所做的选择都是关于一致性和可用性之间的权衡的，而数据共享策略则涉及延迟和一致性之间的权衡。

一方面，你有“恰好一次”的方案，可确保操作执行完成或失败。然而，这也是最不可用的解决方案（参见图 14-9 的左半部分），因为强一致性要求意味着选择的是一致性而放弃可用性。如果出现问题，系统可能在某些情况下为了保证一致性而变得不可用。在天平的另一端，则是较弱的一致性，但具有高可用性。通过接受偶然的请求丢失，你接



受状态或数据的不一致，并在系统语义层面处理这种不一致。因此，你甚至可以在网络分裂的情况下继续服务请求。“至少一次”的方案则是一种妥协，其可确保请求至少在某个节点上成功执行。然后，在必要时，由系统的语义处理节点间状态变化的传播和合并。

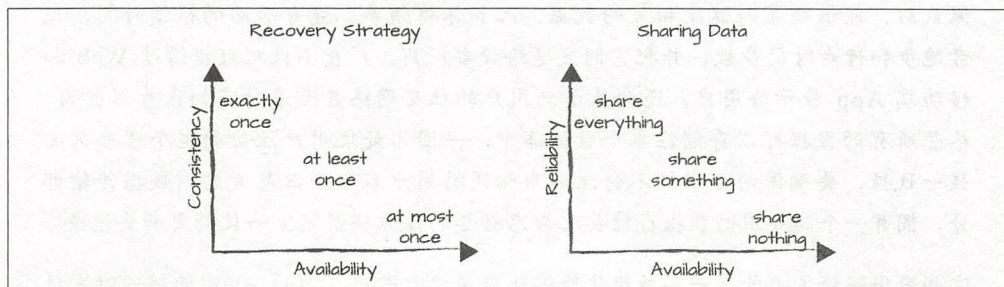


图14-9：一致性、可靠性、可用性之间的权衡

关于数据共享的方法（参见图 14-9 的右半部分）也存在类似的争论，核心是关于可用性和可靠性之间的权衡问题。使用跨节点完全共享的方案，可让你的系统更可靠，因为任何具有数据和状态副本的节点都可以正确接管请求。虽然并不总是可以保证数据被复制，但它是一种比“部分共享”或“不共享”更安全的方案，因为那两种方案会在故障时丢失一部分或者全部数据。

理想的极乐世界是达到两个图表的右上角：一个一致、可靠和可用的系统。在这种情况下，丢失一个节点没关系，因为可以保证状态已经被复制到了至少一个其他节点上，而且请求可以保证要么被精确执行且仅被执行了一次，或者失败并向客户端返回错误消息，这些使你的系统能够始终处于一致的状态。可惜，把这几项全部做到是不可能的。如果可以的话，那么每个人都会选择这样做，分布式系统根本就不会这么难了！

总结

在本章中，我们介绍了可用性的概念，把它定义为系统的正常运行时间，错误和维护时间包含在内。可用性这一术语包含以下这些额外的概念：

- 容错，指让你的系统在出故障时以可预测的方式执行。可能的故障包括进程或节点的丢失、网络连接出错、硬件出错等。
- 弹性，指让你的系统从故障中快速恢复。这包括崩溃后自动重启节点，以及主网络失效后冗余网络的自动接入等。
- 可靠性，指当处于某些预定义好的状况（包括错误状况）时，你的系统依然能够继续发挥功能。如果一个节点由于终止、缓慢或者网络分区而无法回应时，你的业务逻辑要能够把请求转发给其他能够正常运行的节点处理。

容错性、弹性和可靠性以及最终的可用性达到何种程度取决于你是否正确应用了 Erlang/OTP 编程模型以及你在数据共享和恢复策略中所做的选择。这些是我们进入更深层次分布式架构设计的基础。我们在第 13 章中介绍的步骤包括：

1. 将系统功能分割为一系列可管理的、独立的节点。
2. 选择一种分布式架构模式。
3. 为你的节点、节点家族以及集群间的相互通信选择适当的网络协议。
4. 定义清楚节点的接口、状态和数据模型。

接下来你就要挑选你的重试和数据共享策略了：

5. 针对每一个节点的每一个接口函数，需要挑选一种重试策略。

不同的函数需要不同的重试策略。当决定使用最多一次（most once）、至少一次（at least once）或是恰好一次（exactly once）方案时，你需要检查调用链中所有可能的失败场景，包括软件、硬件、网络。使用恰好一次方案时更要特别注意。

6. 针对全部数据和状态，挑选在不同节点家族、不同集群、不同节点类型间共享它们时的合适策略，特别是要把重试策划纳入考量。

就数据共享策略而言，对于状态和数据在节点家族、集群和系统间的共享，你需要决定你需要的是无共享（share nothing）、部分共享（share something），还是完全共享（share everything）。你还可以使用一致性哈希来实现同一份数据的多份副本，并且不需要复制到所有的节点。

在决定选用哪种共享和重试策略时，你可能需要回顾并更改你在步骤 1~4 中所做的设计选项。你可以将各种共享和恢复方案混合，区分不同数据、状态、请求，以进行有针对性的处理。并非所有的请求都必须恰好执行一次，也不是所有数据都需要在所有节点之间共享。“保证交付”“全部共享”之类的方案是昂贵的，所以只将它们用于需要它们的数据和请求的子集。还有别忘了，什么事情都会失败。尝试隔离状态，尽可能不在进程、节点和节点家族间共享。拥抱失败并将其纳入你的架构。尽管诱人，但我们不可能让系统既完全共享数据，同时又能强一致、高可靠以及高可用，在实践中我们必须依据系统需求以及开销做出合适的取舍，以使我们能做到给客户的承诺。

◀ 403

接下来是什么

介绍了分布式架构以及如何基于数据复制和重试策略提高可用性这两个主题后，现在是时候看看可伸缩性了。在下一章中，我们将介绍实现规模伸缩所需权衡的一些问题。我们来看看负载测试和负载调节技术，并了解检测系统瓶颈的方法。



水平规模伸缩

为了伸缩规模而采用分布式，和为了可用性而借助冗余性一样，都是通过在不同的计算机上运行许多个节点实例达到的。然而计算机之间会出现（并且必然出现）失效和失联，所以规模伸缩就不仅仅是增加计算机数量那么简单了。相反，你需要权衡利弊后先选择好系统的一致性与可用性模型，然后在此基础上建立规模伸缩的方案。实现能够无限伸缩而不会丢失任何一个请求的系统——说起来轻松，做起来才会发现绝非易事，而且在实践中其实也罕有真的需要如此理想化系统的应用。也不是说只要用 Erlang/OTP 打造的系统就能魔法般伸缩，但是使用 OTP 确实在很大程度上能帮助你做出许多正确的取舍并避免很多痛苦。

在本章，我们以第 13 章和第 14 章介绍的分布式编程模式、恢复数据、数据共享模式为基础，更进一步把注意力转向架构设计过程中规模伸缩方面的一些权衡问题。我们将介绍一些测试方法，意在使你能够弄清楚你所设计的系统的容量限制，确保其能够无故障地处理请求。这样一来你就可以为你的系统规划好可控边界，使其不会淹没于过量的用户请求中。最后一件你想在重负载下处理的事是节点崩溃、吞吐量降低，或者是第三方服务提供商无响应。

水平规模伸缩与垂直规模伸缩

所谓系统的规模伸缩能力，指的是能够按需应对变化，并保持行为可预测的能力，特别是在高峰和持续重负载的情况下。规模伸缩能力可以是垂直的——使用性能更强劲的计算机；或者是水平的——增加节点和硬件数量。



Amdahl 定律

Amdahl (阿姆达尔) 定律用于预测并行程序内添加内核后的加速程度上限。简单来说, 它告诉我们, 程序将与其最慢的组件一样快。处理并行和并发时, 最慢的组件是你的顺序代码。Amdahl 定律指出, $S(N) = 1/((1-P) + P/N)$, 其中 $S(N)$ 是系统在 N 个核心上执行时能够获得的加速, 而 P 是程序中能够并行的部分的比例。当 N 接近无穷大时, 最大的加速程度变为 $S(N) = 1/1-P$ 。

你可以为你的并行代码投入尽可能多的核心, 但是如果你的顺序代码需要 100ms 来运行, 那么无论你的并行代码运行得有多快, 你都不可能把速度快过 100ms。另一种看待这一规律的方式是: 如果你的代码中有 5% 属于顺序的, 那么你的最大加速上限是 20 倍; 而如果你的代码中有 50% 是顺序的, 你的最大加速上限是 2 倍。在图 15-1 中我们可以看到这一点, 其中还显示出了收益递减的规律。

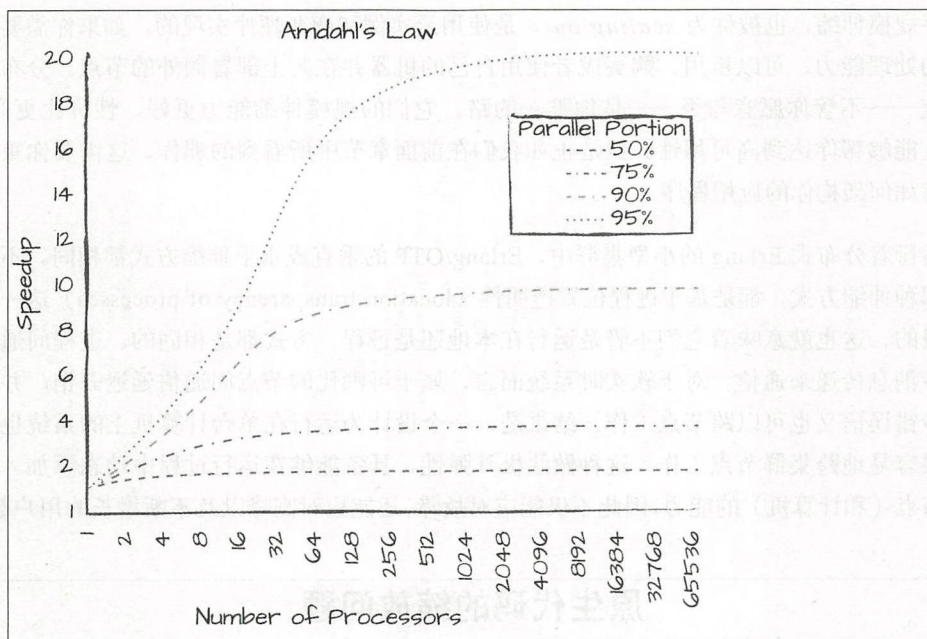


图 15-1: Amdahl 定律

当达到一定的限制时, 添加更多的核只能稍稍提高性能。这时候把你的系统的数据集划分到分布式节点, 使其并行运行就能显现出价值了。

垂直的伸缩, 也称为 *scaling up*, 很容易被接受。你用的是单台服务器, 能够保证数据的强一致性。你只需安上更大的芯片, 更快的时钟周期, 更多的内核和内存, 更快的磁盘,



以及更多的网络接口。想象一下,你打开一个盒子,里面放着的是最快、最闪亮、最高容量,并且尺寸还最小的计算机,哎呀,这种感觉谁不喜欢呢,迫不及待想要马上用它配上你的软件来测测性能了吧?

但是,这种套路已经过时,因为服务器体积就这么大,如果它变得更大,那也就变得更贵。而且别忘了,同样的服务器你最少得买两台,因为虽说它们是超级快的计算机但也还是会存在单点故障。

关于水平规模伸缩方面还存在另一个争论的焦点,那就是多核问题。即使机器能够支持的核心数量可达到几千个,而且先不考虑你的程序本身的并行能力以及是否存在瓶颈,问题在于单个 VM 能够高效使用的核心数量达不到这一规模。不要忘了, Amdahl 定律不仅仅作用于你的 Erlang 程序,也作用于 Erlang VM。这也就意味着,为了能充分发挥硬件性能,你必须在单台计算机上运行多个分布式的 Erlang VM。如果你需要考虑如何在多台计算机上运行多个 Erlang 节点的问题,那么你实际上就是在解决规模伸缩问题。

水平规模伸缩,也被称为 *scaling out*,是使用云实例和商业硬件实现的。如果你需要更多的处理能力,可以租用、购买或者使用自己的机器并在其上部署额外的节点。分布式系统——不管你愿意与否——是你唯一的路。它们的规模伸缩能力更好,性价比更高,并且能够帮你达到高可用性。但是正如我们在前面章节中所看到的那样,这需要你重新思考如何架构你的应用程序。

在运行着分布式 Erlang 的小型集群中, Erlang/OTP 的垂直或水平伸缩方式都相同。不管是哪种伸缩方式,都是基于进程位置透明性 (location transparency of processes) 这一点达成的,这也就意味着它们不管是运行在本地还是远程,方式都是相同的。进程间通过异步消息传递来通信,对于软实时系统而言,属于可消化的节点间通信延迟开销。并且异步错误语义也可以跨节点工作。结果是,一个设计为运行在单台计算机上的系统也能够很容易地跨集群节点工作。这种做法极具弹性,具备能够在运行过程中动态添加/移除节点(和计算机)的能力,因此不仅能应对故障,还能应对高峰以及不断增长的用户数。

原生代码的缩放问题

关于 Erlang/OTP 的一个鲜为人知的事实是它的优越性是体现在整个体系中的每一处的。它支持各种标准网络协议,允许它支持与各类组件进行通信并将它们桥接在一起形成完整应用程序。它还可以通过其设计优良的网络套接字 API 等设施轻松处理专有协议。此外, Erlang/OTP 提供了端口,允许应用程序与外部程序调用和交换数据。开发人员使用这些以及其他 Erlang/OTP 功能成功地构建了数据库驱动程序、JSON 解析器、特定用途的 Web 客户端和服务端以及其他面向集成的组件和

应用程序。

可伸缩的系统通常包含以不同编程语言编写的多个组件，因为不同的语言能够互补优缺点。有时 Erlang/OTP 内置的功能还不够，比如一些应用程序需要大量的数学计算，而 Erlang 不太适合。某些 application 可能需要访问非 Erlang 库，这些库在 Erlang 中重写的难度较高或者成本昂贵等。

由于这些或其他类似的原因，Erlang/OTP 提供了直接从 Erlang 代码调用非 Erlang 函数——称为原生实现函数（NIF，native implemented functions）——的支持。Erlang/OTP 本身的一些部分就是以 NIF 方式编写的，例如 list、map、ets 和 crypto 标准加密模块等。对比其他 Erlang 函数，NIF 看起来类似常规的 Erlang 函数。它们接受常规 Erlang 数据（term）作为参数，并返回 Erlang 数据，但在这些功能内部，实际是以不同的语言（通常为 C 或 C++）实现的。但是，它们是在 Erlang 运行时内直接执行的。当运行时加载包含 NIF 的 Erlang 模块时，它将一同加载包含原生（native）函数实现的共享库，然后使用调用本机函数的指令对模块的 BEAM 代码进行修补（patch）。运行时为 NIF 提供了一套 C API，允许它们访问和创建 Erlang 数据、向其他进程发送消息、引发异常，甚至调度其他 NIF 以供将来执行等。有关 NIF API 的完整描述，请参阅 Erlang/OTP 发行版中附带的 erl_nif 手册页（http://erlang.org/doc/man/erl_nif.html）。

如果经过测量，你发现你的应用程序中的某个部分值得以性能原因重写为 NIF，或者如果你必须重用某个现有的 C/C++ 库而不是在 Erlang 中重新实现它，那么请务必小心，因为行为不佳的 NIF 可能会对 Erlang VM 造成严重破坏。如果你正在写一个 NIF，那么请忘掉“任其崩溃”原则；NIF 代码直接运行在运行时调度器线程上，因此如果 NIF 崩溃，则会导致整个 VM 关闭。你还可以通过使 NIF 一次运行超过 1~2 毫秒来使虚拟机更加阴险和缓慢地死亡，因为这将导致 NIF 占用 VM 调度程序线程并破坏各个调度程序线程间原本精心设计的协同。随着时间的推移，这种中断最终可能导致一种被称为“调度器折叠”（scheduler collapse）的现象，在这种状况下调度器误认为它们没有工作要做，并且错误地进入睡眠状态，只留下一个调度器来处理所有的工作负载。

为避免这种情况，请确保你的 NIF 在快速执行，或者将整个工作分割为短的片段（chunk）然后调度执行的方式，这需要借助 enif_schedule_nif() C API。另一个替代方案是使用 VM “脏调度器”，它们设计来专门用于运行 NIF 和原生代码，不受正常调度器所受的规则约束。脏调度器在 Erlang 17 和 18 中被标记为实验性功能，默认情况下是被关闭的。我们希望在 Erlang 19 中它们会作为常规的 Erlang 运行时功能为各种应用程序所用。

容量规划

明确各种节点类型各需要哪些资源，以及它们如何相互交互，然后以高效低开销为目标确定所需的硬件和基础设施。这样的工作称为容量规划，它的工作的目标是，保证系统能够承载住设计时确定的负载，并且在未来能够按需继续增加规模。

而要想确定负载、确定资源利用率、平衡协同工作过程中对不同种类节点的需求量，唯一的办法就是模拟高负载，以端到端的方式对系统进行测试。这样才能保证众多节点能在高负载下协同工作，并以可预测的方式处理所需的容量，而没有任何瓶颈。它还允许你测试你的系统在出故障时的行为。

在第 13 章中，我们建议把系统功能划分为节点类型和家族，并在集群内连接这些节点。尽管有人争辩说把所有节点类型上不同的应用——前端、业务逻辑、服务功能——都放在同一个节点上可以运行得很快，因为一切都运行在同一个内存空间内，但是这种做法是不推荐的，除非对于简单的系统。对于复杂的系统，应分而治之，毕竟如果节点的功能简单而单一，研究和优化其吞吐量和资源利用率会很容易。

作为一种优化训练，你可以试着去平衡你的系统，也就是尝试减少硬件、操作和维护方面的开销。假想有这样一些前端节点，同一时刻它们能处理的请求量相对较少，但是却面向客户的接口，打开了数以百万计的 TCP 连接。这些节点很可能是内存密集型的，因此硬件需求有所不同，其他 CPU 密集型的前端节点则不需要这么多内存，但是它们的特点是流量更密集，连接更多，花费许多时间在解析和生成 JSON、XML 上。逻辑节点则负责路由请求和运行计算敏感的业务逻辑，它们需要的是更多的核心和内存，而服务节点则负责管理数据库，因此很可能是 I/O 密集型的，需要快速的磁盘。

规划容量时常会忽略的一个问题是，在软件、硬件、网络出故障的情况下，要确保你仍然能够处理设计的负载。如果你的系统中每一个逻辑节点对应两个前端节点，它们的运行时内存和 CPU 都达到 100%，这时失去一个前端节点意味着你将只能处理设计负载的一半。为了确保没有单点故障，你至少需要三个前端节点，每一个运行时最高只能使用 66% 的 CPU 容量；而后端节点需要有两个，平均每个的 CPU 占有率是 50%。这样，丢失任何一个机器或节点都仍然可以保证能够满足需求处理高峰时的负载。如果你希望有三倍的冗余度，那么在这个问题上要投入更多的硬件。

开展容量规划时，你要测量和优化你的系统的吞吐量和延迟。吞吐量指的是通过系统的单元数量。如果请求很统一的话，单元可以按照每秒请求数来算，但是当处理各个请求所需的 CPU 和内存量差别较大（想想 E-mail 或者 E-mail 附件）时，则按照每秒 kilobytes、megabytes、gigabytes 数来算会更好。

延迟指的是服务一个请求时花费的时间。随着负载变化,延迟可能也会变化,并且常常与特定时间点流经系统的并发请求数有关。更多的并发请求通常意味着更高的延迟。

Erlang 运行时系统作为一种平衡的系统在重负载下也能够维持恒定的吞吐率,行为可预测,适合于大多数场景。但是也可能因为出现极端的使用率尖峰或第三方服务响应速度较慢而导致积压请求、Erlang VM 由于系统资源不足而导致中断的情况,以及需要应用负载调整,以使延迟能够维持在预定的范围内的情况。

在第 14 章的“一致性和可用性之间的权衡”一节中,我们根据你的恢复策略与数据共享策略以及分布式架构模式讨论了一致性和可用性之间的权衡问题。你可能还没意识到,那也是在进行可扩展性的权衡(参见图 15-2)。

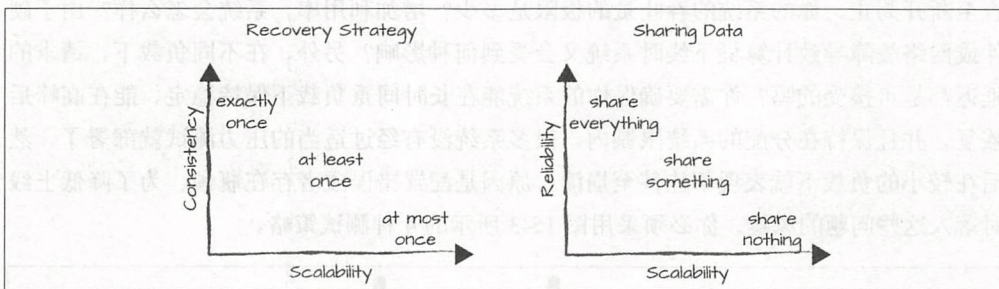


图 15-2: 可伸缩性的权衡

411

伸缩性最强的框架是 SD Erlang。使用它,你可以在一个 `s_group` 内高效地共享数据,但是在各个 `s_group` 之间将共享降至最低。数据和工作流要跨域 `s_group` 共享,需要经过网关节点。通过控制 `s_group` 的尺寸和网关的数量,你可以在单个 `s_group` 内获得强一致性,而在 `s_group` 间获得最终一致。

Riak Core 走的是后一条路,并且虽然它是一个完全网格化的 Erlang 集群,但是它通过使用一致哈希来共享你的数据以及在集群间平衡工作同样提供了伸缩性。你可以把它当作一个运行业务逻辑的巨型交换机,连接起集群中的各个服务节点,但是核心本身又无须完全网格化。核心内连接了上百个节点,每个节点每秒处理数千个请求,即使是最严格的可伸缩事件驱动系统也应该能够归入此类了。感谢 `vnode`,它使得你添加(或删除)节点时能把中断最小化。

最后,一个分布式的 Erlang 集群伸缩性是有限的,但也完全足以适应绝大多数的 Erlang 系统。即使你的目标是每秒处理数万个请求,你也会发现它是绰绰有余的。规划容量时应立足于实际需求,只有在确实有需求时才添加复杂性。

在规模伸缩性的一端是“恰好一次”和“完全共享”的方式,导向的是一致和可靠。

从 CPU 功率和网络需求方面来看，它们也是最昂贵的，因此也是最不可伸缩的。如果你想要一个真正的可伸缩系统，需要把共享的数据量降到最低，并且，如果你不得不共享数据，只要适合就使用最终一致方式。使用异步消息传递来跨节点通信，并且当你需要强一致性时，尽可能把它最小化至少量节点，并把它们放置得尽量靠近以降低网络故障的风险。

412 容量测试

对于开发的可伸缩和高可用系统，必须进行容量测试，以确保其稳定性并了解其在重负载下的行为。这一点总是成立的，与你使用何种编程语言开发系统无关。

直至断开为止，你的系统的吞吐量的极限是多少？增加利用率，系统会怎么样？由于硬件或网络故障导致计算机下线时系统又会受到何种影响？另外，在不同负载下，请求的延迟都是可接受的吗？你需要确保你的系统能在长时间重负载下保持稳定，能在高峰后恢复，并且保持在分配的系统限制内。很多系统没有经过适当的压力测试就部署了，然后在较小的负载下就表现不佳甚至崩溃，原因是配置错误或者存在瓶颈。为了降低上线时落入这些问题的风险，你必须采用图 15-3 所示的 4 种测试策略。

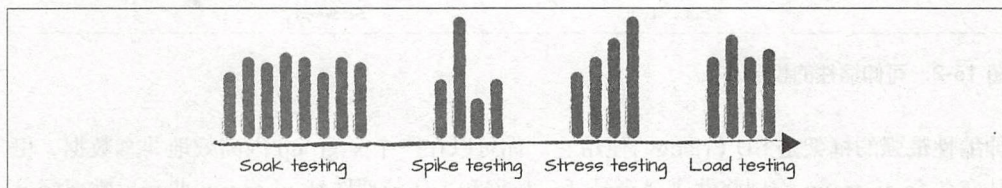


图 15-3: 容量测试策略

它们是：

浸泡测试（soak testing）

随着时间创建持续而一致的负载，确保你的系统能够持续运行，而不存在性能降低的情况。浸泡测试可以持续数月并且不仅测试你的系统，还涵盖整个技术栈和基础设施。

尖峰测试（spike testing）

确保你能够处理尖峰时刻的负载并从中快速而无痛地恢复。

压力测试（stress testing）

逐渐增加生成的负载，直到达到瓶颈和系统限制。瓶颈即系统中的积压，通常表现为长消息队列。系统限制包括耗尽可用端口数、内存，甚至磁盘空间。当你找到瓶

颈并将其移除后，再次运行压力测试继续解决下一个瓶颈和系统限制。

负载测试 (load testing)

以一定的速度让系统接近其极限，确保系统的稳定性和平衡性。负载测试至少要持续运行 24 小时，确保吞吐量和延迟不会出现降低。

不要低估为了消除瓶颈、达到延迟可预测和高可用而所需花费的时间、预算和资源。你需要硬件来生成负载，需要硬件来运行你的模拟器，以及需要硬件来并行运行多个测试。由于产生崩溃需要数天，因此测试必须能够并行而且完全可视化。当你在调试和优化软件栈、硬件、网络设置时，有时候你会感觉仿佛是在大海捞针。

生成负载

对于不同的系统和组织，你可以使用不同的方式生成负载。你可以使用已有的开源工具和框架，比如 Basho Bench、MZBench、Tsung、商业产品，或者 SaaS 的负载测试服务。有些工具允许你记录和回放实时流量。或者如果你想模拟复杂的客户端业务逻辑或者测试简单的场景，自己编写测试可能更简单。你很快会发现，为了测试一套 Erlang 系统，你很可能需要一套 Erlang 编写的负载工具。

如果你连接着第三方服务，或者想针对节点类型进行独立测试，你会需要编写模拟器，因为你的第三方服务不太可能允许你针对线上系统进行测试。模拟器常常是独立的 Erlang 节点，如图 15-4 所示，会暴露出一些外部 API，并在一定的智能程度上重复所模拟的目标的行为。它们被设计为处理你的外部服务的负载，但常常又不止于此。



为开始测试系统的最终实例做上线准备前请务必极其小心，确保你连接上了模拟器，并且把涉及外部服务商的部分流量做了限流。我们建议你最好明确外部服务提供商是否拥有负载控制能力，因为在这方面我们曾犯过错。我们曾经在测试自己写的一个自动拨号器时，忘了将流量转给模拟器。这个错误导致我们原本计划合作的一家 IP 电话提供商出现了重大的中断。他们不太高兴。我们也高兴不起来，因为我们被踢出去，不得不赶在上线前找到另一个新的供应商。

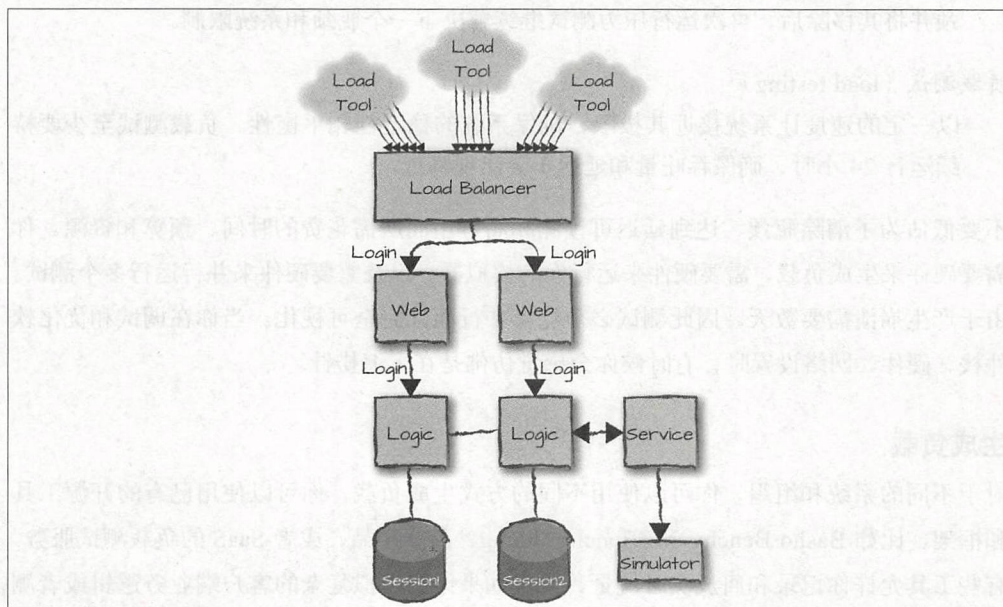


图 15-4: 一套承载了负载的 Erlang 系统

414 平衡你的系统

当恰好平衡的 Erlang 系统满负载运行时，吞吐量应当维持恒定，而延迟会发生变化。假设每个请求的工作开销都是常量，而你的系统峰值是每秒 20 000 个请求，那么任意时刻，当 20 000 个请求正流经你的系统时，峰值延迟应该是 1 秒。如果 40 000 个请求正同时流经系统，则将会消耗 2 秒时间来服务这些请求。因此，虽然吞吐量保持相同——每秒 20 000 个请求——但延迟翻倍了。BEAM VM 是少有的能够具备如此性质的虚拟机之一，即使在极端负载下也能为你的系统提供可预测性。

图 15-5 中展示的是一个未优化过的典型的 Erlang 系统，其中 y 轴代表的是吞吐量。其单位可以是每秒处理的即时消息数，Web server 发送的是以兆字节为单位的数据量，或者是格式化并存入文件的日志条目数。 x 轴展示的则是在同一时刻同时流经系统的请求数。这种吞吐量下降通常是在 CPU 飙高之后出现。在那一刻，流经系统的请求越多，吞吐量越低，而延迟越高。理解 BEAM 虚拟机的这一行为对你来说尤为重要，因为这与你紧密相关。

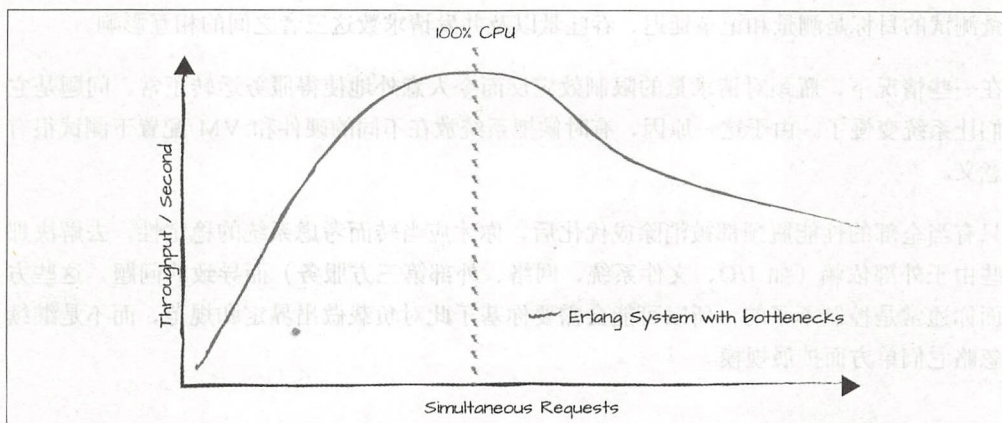


图 15-5: 负载下 Erlang 系统的降级

图 15-6 展示的是移除系统中瓶颈后的结果。不管当前并发请求量有多大，你都能获得稳定的吞吐量。峰值吞吐量会低一点，但这是为了做到无论系统并发请求数量有多大，行为都能够可预测而付出的些许代价。大多数其他语言则会因为进程需要频繁进行高昂的上下文切换而导致吞吐量下降。Erlang 虚拟机针对并发进行了高度优化，使用它能极大地降低你的风险。节点的能力上限受限于 CPU 负载、可用内存，或 I/O。我们把达到这些限制的节点分为 CPU 密集型、内存密集型、I/O 密集型三类。交叉区域反映出了不平衡的系统性能降低。

415

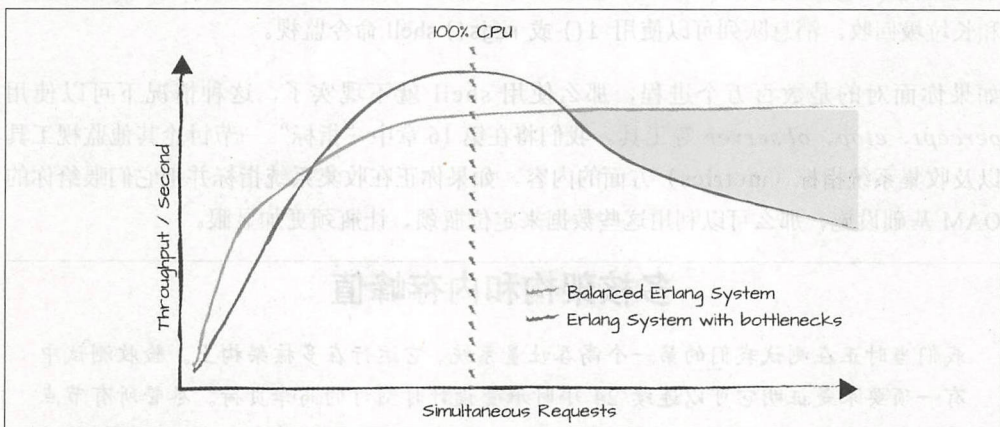


图 15-6: 调整后的 Erlang 系统能应对大负载

为了找到系统的瓶颈，请测试单一节点。使用模拟器，但是要避免过早优化。有些节点类型可能不需要优化，因为它们可能永远都不会碰到重负载。并且你可能会遇到一些原本超快的节点却由于调用外部 API 时服务级协议太慢而成为瓶颈的情况。你进行各种容

416

量测试的目标是测量和记录延迟、吞吐量以及并发请求数这三者之间的相互影响。

在一些情况下，瓶颈对请求量的限制效应反而令人意外地使得服务运转正常。问题是它们让系统变慢了。由于这一原因，有时候把系统放在不同的硬件和 VM 配置下测试很有意义。

只有当全部的性能瓶颈都被消除或优化后，你才应当转而考虑系统的稳定性，去解决那些由于外部依赖（如 I/O、文件系统、网络、外部第三方服务）而导致的问题。这些方面你通常是控制不了的，所以可能会需要你基于此对负载做出界定和规范，而不是继续忽略它们单方面扩展规模。

找寻瓶颈

当你要找寻进程和节点上的瓶颈时，通过监视进程内存使用情况以及邮箱队列就能轻松地发现大多数问题。使用 `erlang:memory()` BIF 可以监视内存用量，它会返回一个元组构成的列表，其中包含动态分配给进程、ETS 表、二进制堆、原子、代码和其他部分的内存信息。

在负载测试过程中，你需要监视不同类别内存的使用情况，确保长期运行过程中资源的使用量稳定，没有泄漏。如果你看到原子表或者二进制堆随着时间尺寸变大，而非稳定，那么如果你持续运行几天、几周或者几个月就会碰到问题。在有些情况下，你需要使用系统监视器（参见第 13 章的“系统监视器”部分），它有助于优化和移除内存消耗高峰和长垃圾回收。消息队列可以使用 `i()` 或 `regs()` shell 命令监视。

如果你面对的是数百万个进程，那么使用 shell 就不现实了，这种情况下可以使用 *percept*、*etop*、*observer* 等工具。我们将在第 16 章中“指标”一节讨论其他监视工具以及收集系统指标（metrics）方面的内容。如果你正在收集系统指标并把它们喂给你的 OAM 基础设施，那么可以利用这些数据来定位瓶颈，让瓶颈更加显眼。

多核架构和内存峰值

我们当时正在测试我们的第一个高吞吐量系统，它运行在多核架构上。验收测试中有一项要求是证明它可以连续 24 小时承受设计时制订的高峰负荷。尽管所有节点都只占用了 50% 的 CPU，而且具有足够的内存空间，但是其中一个用于衔接第三方服务提供商 API 的节点平均每 8 小时就会崩溃一次。于是我们根据服务级别约定限制了请求量，确保同时发出的请求数不超过几百个。并且每 10 秒钟轮询一次内存，每次崩溃之前都会读取数百兆字节的可用内存。然后重写代码，将内存消耗和 CPU 负载降低了 50%，然而都只是延迟了问题，而不是消除它。该节点现在

每 16~20 小时崩溃一次。

最终我们打开了系统监视器，并注意到，在崩溃前几秒之内，产生了非常多的长垃圾回收和大量的堆跟踪事件。这是由于连接到会话创建时会发送回一个 XML 文件，其中包含了会话数据，解析它们导致了巨大的内存峰值。我们在图表中可以看到这些内存尖峰，但是并没有想太多，因为这些峰值尚处于可接受范围内。而每次崩溃发生前都伴随着初始化会话请求的激增，导致这些峰值聚集在一起，并创建了一个导致 VM 内存不足的怪物尖峰。我们最终发现使用更多的核心增加了这个怪物发生的可能性。在不到半秒的时间内，这种内存浪涌耗尽了所有可用的内存，并导致节点崩溃。

那解决方案是什么？我们为会话初始化处理创建了一个单独的 FIFO 队列，以节制请求数量避免过量内存使用。虽然我们控制问题的解决方法是添加瓶颈，但是内存图形变得平缓了，并且吞吐量没有受到影响，系统最终通过了压力测试。

然而很多时候最大的挑战却不在于寻找瓶颈，而是如何才能创建出足够的负载让你的系统能暴露出那些瓶颈。多核架构让这一问题变得更加困难，因为巨大的负载下暴露出的问题常常不是 Erlang 方面的问题，反而是底层硬件、操作系统、基础设施等栈中其他方面的问题。通过使用 `erl +S` 标签，可以让 Erlang 虚拟机在更少的核心上运行，这样相当于削弱了节点的硬件性能，对于检测某些瓶颈有帮助。

同步调用与异步调用

就大多数情况而言，瓶颈的特征之一是长消息队列。假想有这样一个进程，它的任务是格式化和存储日志到文件。假设每处理一条请求，我们需要存储几十条日志。然后我们开始使用 `gen_server:cast` 异步发送日志请求给一个日志服务器进程，但该日志服务器进程是无法承担这样的负载的，原因是请求的速度太快，超过了它能处理的速度。而如果在这种情况下，生产者多达数以千计，文件 I/O 又不够快，那么你将看到日志服务进程的邮箱变得非常巨大。这一队列就是瓶颈对你的系统产生的负面影响的在外表现。为什么会发生这种情况？

你的程序的每一个操作都被赋予了一定数量的余量（在第 2 章的“多核、调度器和余量”部分有介绍），每个操作大致相当于一次 Erlang 函数调用。当调度器调度进程时，每个进程都被指定了一定数量的余量可供执行，每执行一个操作，就会减少一定的余量。当进程到达某个 `receive` 分句处、对邮箱中的消息进行匹配而一无所获时，或者是当该进程的余量减为零时，进程就会被挂起。当进程邮箱的尺寸增长，Erlang 虚拟机会通过增加向该进程发送消息所需的余量数来惩罚发送方进程。之所以这么做是为了控制住生产

418

者,使得消费者能撑住。这种设计方式给予消费者一个机会能够在高峰后赶上,但是如果持续处于重负载,这种做法会对系统的整体吞吐量产生影响。然而这一场景暗含的假设是,不存在瓶颈。对于重负载情况,通过添加余量的方式惩罚发送方并不足以阻止消息队列的增长。

为了能够调整负载和控制流量,以及为了消除这些瓶颈,一个技巧是即使不需要服务器的回应,也使用同步调用。当你使用异步调用时,生产者在前一个发送的请求收到确认前不会再发送一个新请求。同步调用阻塞了生产者,直到消费者已经处理了前一个请求,这样也就防止了邮箱被塞满。这会有和图 15-6 中所示一样的效果,以牺牲吞吐量为代价,获得稳定和可预测的系统。使用这种方式,记得要调优超时值,不要认为默认的 5 秒就是合适的,也绝对不能把它设置为无限等待(infinity)。

减少瓶颈的另一种策略是尝试降低消费者的工作负担,如果可能应把一些工作量转嫁给客户端。在记录日志这个例子里,与其逐条处理,不如试试进行批处理,积攒几百条后一次写入磁盘。你还可以把工作推给发起请求的那些进程,比如把格式化工作让它们来完成,而不是都推给服务器。毕竟,格式化日志是可以并行进行的,而写入日志却必须顺序进行。

现在你已经优化了代码,认识到了系统的极限,以及了解了定位瓶颈的方法,然后你需要保证的是,即使达到这些极限,你的系统也不会出故障或者性能下降。

系统蓝图

如果你已经达到这一步,现在是时候把你的设计决策升华为集群蓝图和资源蓝图了,然后把这二者组成系统蓝图。资源蓝图指的是你的集群运行时可用的资源。这包括对硬件规格、云实例、路由器、负载均衡器、防火墙,以及其他网络组件等的描述。

集群蓝图则衍生自你在容量规划中所了解的内容。它是对你的系统的逻辑描述,指明了各个节点家族,以及它们相互之间的连接关系。你还定义了不同节点类型之间的数量比例关系,遵循这一比例的系统是平衡的,能够正常运作而不会降低服务质量。你的编排程序可以使用此蓝图,以确保集群可以有序地被缩放,而不会在节点间造成不平衡。它还保证了你的系统在出现故障后能够持续运行,而不会降低服务质量。集群蓝图有点类似于 Amazon Web Services 上的 Auto Scaling Groups,但是细节更加丰富。当你的一个集群达到极限时,就部署新的集群。

你的集群蓝图和资源蓝图合在一起构成的就是我们所谓的系统蓝图。有系统蓝图在手,你可以知道你的分布式系统的结构是怎样的,以及如何将其部署到硬件或云实例上去。

负载调节与背压

很久很久以前，在一个遥远国家的一个新年夜，大家都拿起电话，相互致电祝愿新年快乐。核心通信线路不堪重负卡住了。但拨打电话依然是可以的，网络尽管存在波动但继续运转。它在当初设计的最大负载情况下行为依然可以预测。

系统因为采用了背压（backpressure）来限制同一时刻主干线路上连接的通话数量上限，因此保持了平稳运行。你总是可以听得见拨号音并且也被允许拨号，但是如果你尝试在没有空闲线路的情况下访问国际主干线路，你的拨打会被以忙音拒绝。所以你可以不断尝试直至拨通。背压是一种机制，它告诉发送方停止发送，因为已经没有容纳新消息的空间了。

技术不断发展，在电话之后又出现了 SMS。随着 SMS 的流行，新年夜的 SMS 使用峰值越来越高，当然投递 SMS 的延迟也相应变大。一旦手机允许你向几十个用户同时发送 SMS，延迟就更糟糕了，消息常常是在凌晨才能到，而发送者（以及接受者）早已等不及上床睡觉了。SMS 很少会被拒绝——它们会通过，但是延迟很多。移动运营商运用的是一种被称为负载调节（load regulation）的技术，这种技术会把所有的请求都投递到队列中，确保没有请求丢失。一旦能够处理，消息就会从队列中立刻被取出，然后发送给 SMSC（SMS 中心，SMS Center）。< 420

相互打电话或者发送 SMS 可能已经是过去的事了，但是这些发展自和应用于电信领域的技术在应对大规模伸缩方面却依然有用武之地。相似之处在于，负载调节和背压都能让系统在面对过载时不至于失效，同时维持吞吐量和可预测的延迟。区别在于，负载调节允许你——借助队列和限制并发连接数——保持和记住请求，而背压直接拒绝请求。如果你针对第三方 API 或者服务节点使用负载调节，请记住你所做的一切只是平滑峰值和低谷，保证不会向第三方投递过量的请求。如果还是源源不断收到请求，速度已经远超处理能力，你最终只能停止继续把请求放入队列，而是拒绝。

Little 定律

Little（利特尔）定律是一个等式 $L = \lambda W$ ，其表示队列长度（queue length） L 等于到达速率（arrival rate） λ 乘以响应时间（response time） W 。在大多数基于因特网连接的程序中，队列长度即等待（以及正在）接收服务的客户端请求的数量，到达速率则是每个时间单位内被系统接收和服务的客户端请求数量，响应时间则是服务一个客户请求需要多长时间。重新排列等式中的参数，我们得到 $W = L/\lambda$ ，即：响应时间 = 队列长度 / 到达速率。这表明如果队列长度变长，或者到达速率降低——更准确地说是吞吐量降低——那么响应时间就会上升。

对于在线系统，你无法控制到达速率，但是即使是在重负载状况下它也可以期待为一个常量。你能控制的是队列长度和吞吐量，办法就是应用背压，并消除请求处理路径中的瓶颈。通过控制平衡系统中的队列长度，以及维持常量的到达速率，你就控制了响应时间。要做到在正确的时间维持正确的值以及应用背压，关键在于你必须对系统拥有完全的可视性，能够看到正在发生的事情，并正确测量它们。

让我们使用一下新年除夕夜短信的例子。如果网关收到的文本量超过 SMSC（其将文本转发到移动终端）可以处理的程度，它会将额外的文本排队存储在负载调节应用程序中，并以先进先出（FIFO）的方式按照 SMSC 可以处理的速率投递。这个速率被称为服务级别协定（service-level agreement）。如果短信持续以很快的速率进入，则队列大小必然会达到极限并溢出。当发生这种情况时，网关开始在逻辑节点中单独或批量地通过触发前端节点中的某种形式的背压来拒绝短信，并且不在网关节点中接受它们。这种情况如图 15-7 所示。

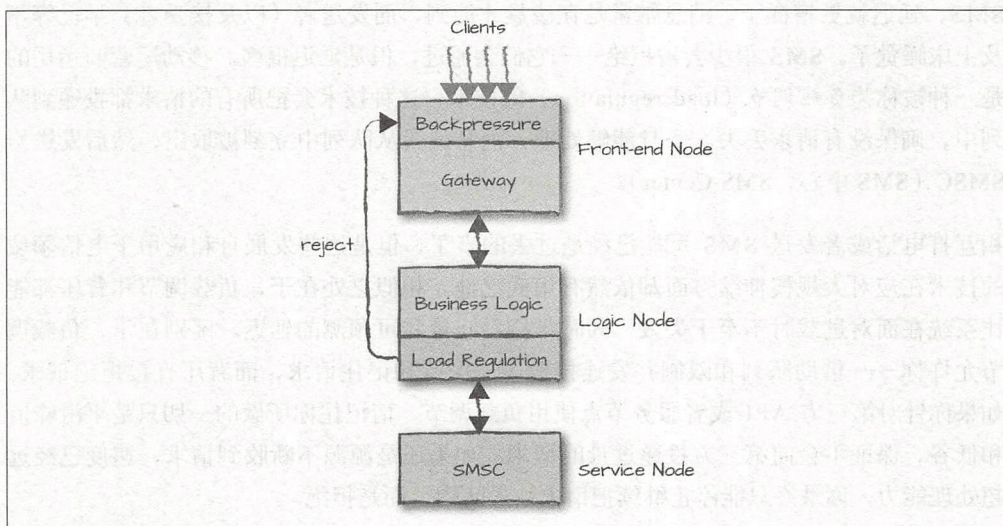


图 15-7：负载调节和背压

为了限制请求和应用背压，你需要使用负载调节框架。这些可以嵌入到你的 Erlang 节点中，或者放置在前端节点和服务节点的边缘部分。控制负载的另一个常见做法是通过负载均衡器（load balancer）。软件/硬件负载均衡器除了平衡前端节点请求外，还可以节制同时连接的用户数并控制入站请求的速率。悲伤的是，这就需要对负载均衡器本身再做压力测试，又打开一罐新的蠕虫。^{注1} 是谁说开发可扩展的弹性系统不难的？

注1 我们经常导致负载均衡器崩溃或行为异常，并且在负载测试时不得不关闭它们，因为它们没有足够强大的能力来承受我们产生的负载。

请记住，负载调节是有成本的，因为你使用了队列，而分发器（dispatcher）可能会成为增加开销的潜在瓶颈。只有当你有需要时才控制负载。当你为本地部署了一个网站后，镇里所有人都同时涌向该网站购买鲜花的风险有多少？但是如果你正在开发的是一个游戏后端，它必须伸缩规模适应百万级玩家，那么负载控制和背压就是必需的了。它们使你能够在面对峰值负载时保持系统的延迟或吞吐量恒定，并确保系统不会因为达到系统限制而降低性能或崩溃。Erlang 里有两个广泛使用的负载控制应用：*Jobs* 和 *Safetyvalve*。

Jobs 与 Safetyvalve

422

Jobs (<https://github.com/uwiger/jobs>) 是由 Ulf Wiger 编写的一个基于 Erlang 系统的负载调节器。它提供了一个排队框架，其中可以为每个队列单独配置吞吐率、作业类型（job type）和并发请求数。你可以在运行过程中动态添加和修改队列。让作业排队以延迟其执行，并限制同时运行的进程数量。*Jobs* application 还允许你为队列中的作业指定超时值，提供了诸如 FIFO（先进先出）和 LIFO（后进先出）之类的策略来从队列中提取作业，并可以限制队列长度。一旦达到队列的长度限制，后续的作业将被拒绝，直到队列中再次出现空间。

Jobs 的调度器实现负载调节的方式符合 Erlang 理念，它为每一个作业分裂出一个单独的作业进程并请求执行许可。若获得许可，它执行完任务并终止。*Jobs* 还能通过采样底层内存使用情况及 Mnesia 负载状况，告知调度器在超过某些阈值时抑制（减少）作业调度速率或并发请求数。采样是基于插件机制的，因此你可以编写自己的插件并检查其他项目，如内存。一旦采样值恢复正常，就会取消抑制。在分布式系统中，*Jobs* 将跨多个节点传播负载状态，从而使各个节点也可以采取适当的措施。

另一个受欢迎的负载调节框架是 *Safetyvalve* (<https://github.com/jlouis/safetyvalve>)。它的灵感来自 *Jobs*，但其专注于更小的领域——排队机制，通过控制吞吐量和允许同时执行的请求数来保护系统避免过载。*Safetyvalve* 允许你配置多个队列。对于每个队列，你可以使用令牌桶（token bucket）算法设置队列类型、队列轮询频率和突发处理等。每次轮询系统时，你都可以向桶中添加令牌。令牌允许你在启动系统时或在不活动时段之后以突发方式执行请求。你可以配置每次轮询后添加令牌的速率，以及令牌桶的最大大小，限制突发的大小，还可以配置队列的大小以及允许执行的并发任务的最大数量。

总结

在这一章中，我们介绍了基于 Erlang/OTP 的系统中与伸缩性有关的一些知识。让你的系

统具有可伸缩性的关键是，要把它设计为由一些松散耦合的节点构成，这些节点能组合并协同运作。这增加了添置计算能力以及按需伸展的弹性。你常常会想在你的节点和节点家族内实施强一致性，但是在别处实施最终一致性。

423 > 第 13 章和第 14 章介绍的架构系统的步骤包括：

1. 将系统功能分割为一系列可管理的、独立的节点。
2. 选择一种分布式架构模式。
3. 为你的节点间、节点家族间、集群间通信选择合适的网络协议。
4. 定义好节点的接口、状态和数据模型。
5. 针对你的每个节点中的每一个接口，挑选一种重试策略。
6. 针对全部数据和状态，挑选在不同节点家族、不同集群、不同节点类型间共享它们时的合适策略，特别是要把重试策略纳入考量。

迭代上述步骤，并在这一过程中反复权衡直至能够符合你的需求。你还会做出一系列选择，这些选择将直接影响到可伸缩性，从而使你必须在可伸缩性、一致性和可用性之间做出取舍。现在：

7. 设计你的系统蓝图，留意系统规模扩展 / 收缩时各类节点的比例关系。

要定义集群蓝图和资源蓝图，你需要明确如何平衡前端节点、逻辑节点和服务节点，这依赖于你所选择的分布式架构模式以及目标平台的硬件情况。你要记住目标——无单点故障——保证你拥有足够的资源能够满足延迟和吞吐量方面的需求，并且即使每种类型的节点都有一个出了故障，你还是能够弹性恢复。当你完成这些后，把它们加入你的系统蓝图。

检验系统蓝图的唯一方式就是在目标硬件上做容量测试。编写模拟程序并执行浸泡 (soak)、压力 (stress)、尖峰 (spike) 和负载 (load) 测试，以消除瓶颈并验证你的假设。

8. 分辨出哪些地方需要应用背压和负载调节。

当对你的系统做容量测试时，应当尽力看清系统存在的局限性。弄明白要在哪里应用负载调节和背压，以保护你的系统不会降低性能或者崩溃。在等待时间过长或某些节点内存不足之前，有多少个并发请求可以通过系统？你的系统是否能够应对故障，避免服务降级？另外，请确保你不会把第三方 API 和服务弄崩溃，遵守协议中约定的服务级别。

最后一点建议是，不要过度设计你的系统。过早优化是万恶之源。不要假设你需要一个分布式的框架，不要仅仅因为存在这样的框架或者你能够使用这样的框架就去采用它。

即使你要编写的是下一代 MMOG（大型多人在线游戏）引擎或者下一个 WhatsApp，也应当从小开始，确保你的系统自始至终能运转。准备好使用这些框架，但是把它们隐藏在软件的 API 抽象层之后，这样你可以在之后某天改变你的策略。另外，在对系统进行压力测试时，记得重建错误场景。干掉节点，关掉计算机，拔掉网线，然后看看你的系统会如何行事，如何从故障中恢复。在这个阶段里你可以决定在可用性、一致性和可伸缩性方面你要做出怎样的取舍。在不丢失任何一个请求和偶尔丢失一个请求的两种系统之间，对基础设施的开销就硬件容量而言可以达到一个数量级的差异。你真的需要承担 10 倍的硬件花销以及接受因此带来的复杂性，为的只是搭建一个没人付钱、很少出错的服务吗？

接下来是什么

现在你已经拥有一个你认为是规模可伸缩、可靠的系统了，你需要确保在系统上线后，你的 DevOps 团队能够可视化地观察系统中正在发生的事。在下一章，我们将介绍指标（metric）、日志（log）和警报（alarm），它们的存在让员工有机会监视系统，并在需要的时候给予支持和维护，并且能在问题出现、恶化和失去控制前采取行动。

监视与抢救性支持

一直读到这里的你，一定已经摩拳擦掌准备打造出一套可伸缩、高可靠并且高可用的系统，令众人钦佩一番了。基于正确的工具和手段，原本只有电信系统才能达到的五个九级别的可用性，如今在你开发的软件中也有望能够达到了。但是要达到这一目标，即使实现了前面章节介绍的全部内容还是不够。与弹性软件、冗余硬件、网络、电源和多个数据中心一样重要，要想实现高可用性，秘诀在于——能够从较高层面可视化地了解系统中发生的情况，以及根据收集到的信息采取合适的行动。

你的 DevOps 团队基于这些信息可以做到：抢救性支持（preemptive support）和验尸调试（postmortem debugging）。通过对系统进行监视，他们可以察觉到故障迹象，并在失控前定位出问题，这一过程既可以是手动的，也可以是自动的。磁盘正被填满？那就触发一个脚本，通过删除旧日志来进行清理。过去几个月，由于注册用户量和并发会话（session）量的增长，系统负载也稳定增长？那就部署更多的节点应对负载，避免容量触底。

无论你是一个多乐观的人，都不可能提前预知所有的问题和错误，除非等到它们开始显现。问题源源不断，逼着你通过更高层次的容错来应对故障。当进程或节点自动重启时，你会希望获得崩溃前一刻系统状态的快照。这份快照加上其他历史数据，使你能够快速而高效地完成验尸调试，明确崩溃原因，并确保未来不再出现同样的问题。

反之，如果缺乏快照，想要有条不紊地进行调试就很难了，常常得靠猜。这简直是逼自己大海捞针。最后还要提醒你，你大概会指望错误会在你坐在计算机屏幕前时，礼貌地显现。它们不会。难道说系统要崩溃时，还要考虑你此刻正在吃饭／喝茶／回家，然后等你？所以，确保你能够可视化地看到数据，并能记录下历史数据，这些额外工作非常值得花时间做好，等你需要找寻错误原因、修复故障，以及安放抢救性指标来确保你碰到的问题不会再次发生时，你就会领悟这背后的价值。在本章，我们将介绍监视（monitoring）、抢救性支持（preemptive support）方面的方案，还将介绍一些常见的自动化支持方案。

监视

借助崩溃转储报告提供的信息，任何人都能明白原来是虚拟机内存耗尽了。但是消耗的究竟是哪一种内存呢？是原子表、程序代码内存、进程内存、二进制堆，还是系统内存？也许是系统遇到了一大拨儿登录请求导致内存消耗出现尖峰？或者是由于缓慢的第三方 API 导致请求延迟升高，使得进程生存周期变长？又或者是某种特定类型的请求失败，触发了 I/O 敏感的清除过程，这又进一步触发了许多其他预料之外的事件或超时。在缺乏可视化手段的情况下，你只能猜测系统的当前状态，而无法在问题恶化前根据趋势定位出问题。而当问题恶化后，由于缺少历史数据，又让排障变成了耗时和令人畏惧的工作。这就是为什么我们要监视系统，并存储信息供之后访问。

要想实现监视，需要组合以下几种基本手段：

- 日志 (log) 记录了程序中的状态变化。包括业务逻辑状态改变，如用户登录和会话初始化这类，或系统状态更改，如节点加入群集。
- 指标 (metric) 是通过在特定时间点轮询值而获得的值。你可以监视系统指标，例如 CPU 利用率、内存使用情况、ETS 表大小、打开的 TCP 连接数，或业务指标，如延迟、活动会话数或每小时登录请求数等。
- 警报 (alarm) 是与状态有关的一种事件。当符合某些标准时，它们会被触发，例如磁盘空间不足或达到并发会话阈值。类似的，当这些标准不再满足时，它们被清除，例如，在压缩或删除文件之后，或者在用户注销之后。

监视功能应当与你系统的配置和管理功能衔接起来一起开发。我们把这些功能分别称为运行 (operations)、管理 (administration) 和维护 (maintenance)，合称为 OAM 系统——或者如果缺少能够配置和管理业务逻辑的部分的话就叫 O&M 系统。在本章的后面，我们会聚焦于监视领域，并且会使用 OAM 术语来指代。

在一些架构师——他们从来没支持过在线 (live) 系统——设计的 Erlang 系统中，OAM 部分常常被遗漏或者是不完整的、勉强补上的。如果你碰到这样的系统，为了能够统计在线会话数，只能手工把所有节点上的 ETS 会话表的长度累加；或者要在线更改配置，得调用 `application:set_env` 才行，那么这样的系统是错的。所有合格的系统都必须具备让人检查、管理和排障的功能，并且使用者无须具有任何 Erlang 知识就能使用这些功能，也不需要访问 Erlang shell。

在电信领域，非紧急的 OAM 功能都会放到单独的节点上（或者节点对上，考虑到冗余的话），原因与我们在第 13 章的“节点类型与家族”一节讨论过的相同，即减少前端节点、逻辑节点和服务节点负担的同时增加弹性。OAM 节点必须被设计为，即使出现故障，系统也应当能够正常服务请求。这就意味着，只有极关键的 OAM 功能才可放置于

非 OAM 节点内,通常只包含极少的几种紧急警报类型,以及能够检测节点存活性的功能。

OAM 节点既能与软件中的 Erlang 组件配合,也能和那些非 Erlang 组件配合。这些 OAM 节点扮演枢纽设施的角色,为更宏观的运维基础设施提供支撑(参见图 16-1)。这里说的更宏观的基础设施,它能够监视和管理你的网络、交换机、负载均衡设备、防火墙、硬件、操作系统以及这整套技术栈本身。其中可能涉及 Graphite、Cacti、Nagios、Chef、Capistrano 等开源工具;或者一些商业工具;或者使用 SaaS 提供商,如 Splunk、Loggly、NewRelic 等。各个部分相互连接时可能会涉及许多的标准和协议,包括 SNMP 和标准 MIB (standard management information bases)、YANG/NETCONF、REST、Web Socket 或者其他近来流行的东西(只要不是 CORBA 就好)。

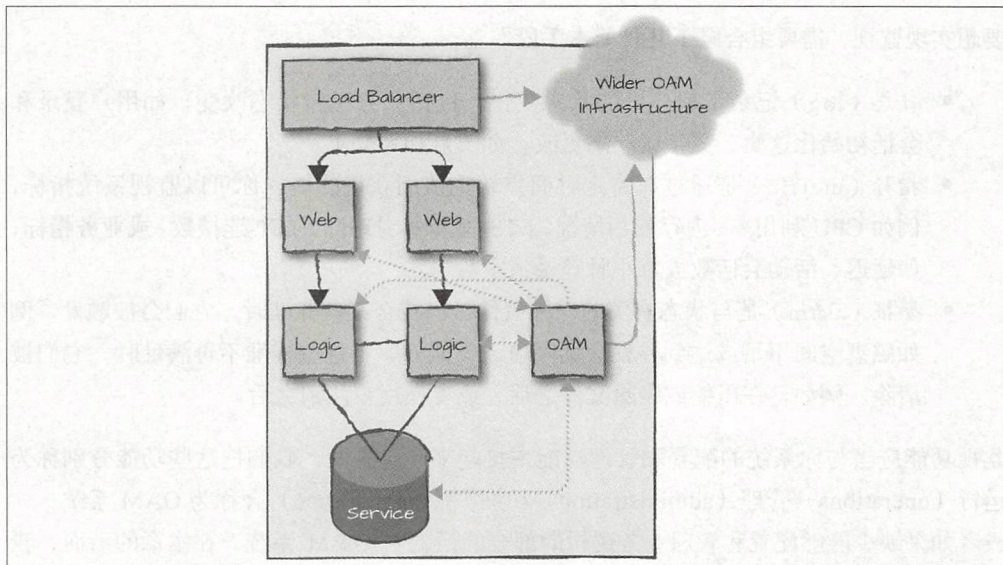


图 16-1: 操作与维护节点

日志

一条日志指的是文件或者数据库中的一个条目,其本身是对某个事件的记录,能够用于审计追踪 (audit trail)。这些条目反映了 Erlang VM、操作系统中发生的事件,或者反映了业务逻辑中发生的状态改变。日志的用途广泛,包括追踪、调试、审计、合规监控以及计费等。不同的日志条目通常被打上不同的标签,使你可以在运行过程中决定日志存储的粒度级别。常见的标签包括 *debug*、*info*、*notice*、*warning*、*error* 等。

由于不同的人掌握的技术技能不同,使用的工具不同,记录日志的方式也不同,所以很难做出“均码式”建议。重点是,为了能够通过日志完成问题定位、数据收集等任务,日志

所提供的信息必须足够完善，使得能够在跨节点环境下追踪任何一个请求的完整流程。

想想我们的电子商务的例子，每天都会有数以百万的请求通过该系统。你如何处理这样的客户投诉——他们抱怨包裹一直没收到，但是信用卡却已经扣了款？你如何缩小搜索范围，并明确消息丢失的原因到底是进程崩溃造成的，还是网络出错造成的？你需要快速找出请求到底是在你代码中的哪个地方消失的，然后承认自己的过错，或者通过可靠的审计追踪证明自己的清白，然后把问题的焦点转向仓库管理团队或者是快递。

你能不能基于客户购买商品的日志进行客户画像（customer profile），或者通过研究用户会话时间特征理解他们的购物行为模式？弄清楚究竟有多少顾客只是把商品放入了购物车却未结账？或者计算出销售总额，然后与处理信用卡交易的银行提供的数据进行比较，以实现收入保证。这些就是我们所聚焦的粒度级别。

429

我们曾经见过 SASL 日志，那是在第 9 章的“SASL application”一节，SASL 是 OTP 自带的，是免费的。如果配置正确，你就能得到一些二进制日志，其中记录了监督者、进度、错误和崩溃等方面的报告。你还能添加自己的事件处理器，把崩溃和错误日志转给某个中心处理。但是保存这些信息仅仅是深入挖掘的开始，你还能发现更多更深的用途。想象这样一个系统，它有数百（甚至数千）个节点，能够弹性地进行伸展和收缩。如果你没有自动把各个节点的日志信息集中到某个中央仓库，你就得自己 SSH 连接到某个机器，然后连接 Erlang shell，启动报告浏览器，搜索崩溃报告，并在内心祈祷其尚未被轮换覆盖。在这样一种状态下，出了问题你怎么能找得出来呢？

我们曾经有一个站点，其上运行的系统中有些进程每天都会崩溃。因为这些崩溃了的进程每天都会自动重启，所以我们并没有察觉到问题，还以为系统运转正常，但实际上请求中总有一小部分是失败的。但故障被隔离得如此之好，我们都不知道系统正忍受着 bug 之苦。这真的很糟。如果你想要高可用性，那么需要自动发现 SASL 崩溃和错误报告，然后确定定位出这些出故障的部分。尽管它们出现的次数看起来不多，但是任何一个面对它们的客户都因此而体验到挫折。而且如果这些问题持续快速出现多次，很可能导致重启次数达到它们的监督者设定的上限，最终恶化到使整个节点都失效。修改监督者配置，增加允许的重启次数之类的做法不是解决办法。你需要追本溯源，从根本上解决问题。

用户常常将一些自己的日志条目添加到 SASL 日志里，但我们不建议这么做，因为这把不同类型、不同用途的日志混在了同一个文件里。如果系统规模比较小，流量不大，这么做可能还可行，但是当你需要每秒处理成千上万条请求，而每条请求都会产生多条日志条目，很快就会耗尽 SASL 日志容量，然后你绝对会想把不同的日志类型分别存储到不同的文件中去。



Lager

Lager(<https://github.com/basho/lager>)是Erlang最受欢迎的开源日志框架之一。它为Erlang系统提供了高度优化的日志能力,并与传统的UNIX日志记录工具(如*logrotate*和*syslog*)相集成。不同的日志级别:*debug*(调试)、*info*(信息)、*notice*(提醒)、*warning*(警告)、*error*(错误)、*critical*(关键)、*alert*(警报)和*emergency*(紧急)可以被分配不同的处理程序,从而允许你决定如何管理这些信息。默认处理程序会格式化你的日志以供离线观看、终端输出,以及转发到短信、寻呼机和其他服务提供商等。大多数OTP错误消息被重新格式化为更可读的消息。*Lager*还具有过载保护和限流功能,允许发送日志并根据邮箱大小在异步式和同步式调用之间切换。它还引入了接收器(sink)的概念,允许你只转发最关键的日志条目。

为了弄清楚该记录哪些与业务相关的日志,可以追踪每个请求的功能信息流(functional information flow),识别出其中哪些可能改变系统状态,并记录下哪些地方可能走向不同的分支。以这种方式考虑将会给维护人员、支持工程师、DevOps团队、会计、审计人员、市场人员、客服人员等提供一个非常好的概览,让他们了解到正在发生什么、已经发生了什么。每当发生显著的状态改变时,记录下一些之前没存储的有用信息。

确保你能够通过一些唯一的标识符,把各个日志条目连接起来,重建出功能信息流。你不能仅仅凭借时间戳来实现这一点,因为数据数量可能很大。你也不能依赖于session ID、user ID或者电话号码之类的数据,因为它们在请求级别上不是唯一的,多个不同请求可能具有这些项的相同的值。正确的做法是,每当外部客户端发送来一条请求时就为该请求分配一个唯一的ID。由于一条来自外部的请求,在系统内部处理时可能又会引发多个对内/对外的独立的请求,所以,各个日志里会出现不同的唯一标识符。为了能将它们衔接在一起,你需要在调用日志函数并存储它们时,便确保生成好这些唯一标识并传入。虽说日志记录可以在之后添加,但是你必须编码前就通盘考虑好日志记录策略,因为你的业务逻辑中用到“唯一ID”的唯一理由可能就是为了日志记录。你不会希望由于考虑失误而必须重构全部代码——直到需要调用一个外部API时你才发现,你没有请求ID,而它本应在某处生成。你还得准备好根据维护人员、DevOps团队以及其他日志消费者的反馈改变记录的日志。

试着减少日志间的重复信息。信息只需存储一次,然后通过标识符与其他日志连接起来。“使用单个日志来存储一切”这种做法只在开发阶段行得通,一旦系统每秒持续产生数万条事务,要想高效地从文件中抽取出有用的数据就很难了,除此之外这种做法还很容易成为瓶颈。理想而言,你的日志应当创建出一种关系模型——根据当前的流,某个文件中的一条日志条目可以通过一个唯一ID与另一个文件中的某个日志条目连接起来。该唯一ID可以是session ID,这就把用户浏览过的商品、在购物车放入/移除的商品,以及结算并支付的商品等连接了起来。可以用一个日志文件记录用户浏览过的所有商品,包括花费在浏览商品上的时间;用另一个日志文件记录其在购物车里添加和移除的商品;

第三个日志文件用于记录购买了的商品。对于结账时付款了的那些商品，则应该有额外的唯一 ID，这些唯一 ID 来自支付网关，把 session 和付款连接了起来。所有的支付日志，相应的也就不必存储 session ID 了，因为在支付日志中已经把这二者连接起来了。

另一种看待日志的角度是把它看作 FSM（有限状态机），其中每个条目代表一个状态，而状态的转移是基于业务逻辑中定义的一系列条件判断做出的。能够回放 FSM 中的状态转移将使得 DevOps 工程师能够复现出用户把商品添加到购物车然后支付的整个过程。

区分不同日志的级别，特别是哪些内容只是在调试模式下会用到，这样可以避免增加系统负担去产生许多无用的日志。就最低限度而言，务必要合理地记录下进入（incoming）和发出（outgoing）的请求，以及它们对应的结果，这样一来你才能在之后找出存在问题的系统和组件。举一个例子，如果你正在调用外部 API，就创建了一个新的日志条目，记录下请求、延迟以及提供给你的唯一请求 ID 等。如果这个对外部服务的 API 的请求超时了，则把结果替换为超时标签。你可以在之后分析这些日志，并考虑是否需要增加超时值。

你必须在与业务相关的日志中记录系统相关的项，而不考虑节点或系统所做的。我们已经讨论过 SASL 日志，这是 OTP 本身的一部分。永远记录下所有的 Erlang shell 命令和交互。你将吃惊地发现有多少次故障中断是由于操作人员缺乏 shell 操作经验导致的。你能知道他们都做了哪些操作，这会变得非常重要，特别是当你需要恢复服务时，可以证明自己的代码没问题。如何记录下 shell 命令，这就和架构有关了。可以通过重定向 I/O、使用 bash 命令、在 Erlang VM 中添加钩子等方式来记录。

其他需要记录的内容包括网络连接，以及内存问题，可以使用 system_monitor BIF 来获得这些方面的通知，正如我们在第 13 章的“系统监视器”部分所描述的那样。不过使用后一种方式要小心，我们曾经经历过这样的状况——由于 24 小时内记录了数百万条 Erlang 端口（port）繁忙通知，耗尽了磁盘空间，最终导致节点崩溃。编写代码时还需要小心的情况包括，避免导致内存尖峰，避免触发长耗时的垃圾回收，避免发送消耗大块堆内存的消息。在这些情况下，使用一个计数器，每次出现这样的消息就增加 1，要比把整条消息记录下来好得多。你可以把这样的功能加入其中，即在是否允许记录消息这一点上允许切换设定，这样在需要调试的情况下就能够获取到细节信息了。

◀ 432

当发生代码被加载、移除、节点重启、成功产生（以及重命名）崩溃转储等情况时，也都值得进行记录。即使是那些你认为不会出问题的事也要坚持记录下来，这会让你活得轻松一点，当异常行为出现，系统进入错误状态时你才能够明确是哪些事件的发生触发了这样的状况。

如果你把日志存储在本地，使用 append-only 式文件无论是把日志以离线方式存储到文

件里、存储到数据库里，或是按 SaaS 提供商提供的方式存储都是很常见的。磁盘空间是廉价的。把日志存储为 CSV 之类的纯文本格式的话就可以导入各种系统，用于故障、计费、合规（compliance）以及收入保障（revenue assurance）。日志可以遵循某种标准，如 *syslog*，或者其他专有的格式。站在使用数据者的角度考虑，要确保数据的格式能被他们容易地使用。这里引用 Pat Helland 的名言：“数据库是事件日志的缓存”，如果你的数据库（或者系统状态）出错了，那么通过日志可以找出原因所在。反之如果没出错，日志也能告诉你为什么没出错。

我的短信在哪里

我们曾经收到一个投诉电话，一个非常愤怒的用户抱怨，他在足球比赛结束几小时后才收到短信通知。根据他的号码，我们发现前端节点发送的短信请求确实到达了我们的系统。在日志中，我们发现当天有三条短信发送到该用户的号码。每条短信进入系统时都被分配了唯一的标识符。

根据日志中的时间戳，我们发现在比赛期间发送了两条球队得分的短信，而第三条则是包含最终比分的短信，我们猜测是在比赛结束后发送的。当短信到达我们的系统时，我们创建了一条带标识符的日志，其中还包含短信类型和费用代码（cost code）——指出应向用户收取多少费用——等信息。然后我们借助标识符以及该短信属于付费型短信的特点来查找后续日志记录。在相关业务逻辑中检查后确定该账户处于激活（active）状态，允许收到付费型短信，而且即使存在预付款类订阅，账户中也有足够的资金可用于支付。几秒后我们就找到了目标日志条目，从中可以看到所有的这些都没有问题。事实上用户是后付费类订户（根据使用情况按月收费），而且他的账户也没被暂停，接收付费型短信也是允许的。于是我们根据唯一标识符，进一步检查了将短信发送给 SMSC 服务节点后其中记录的日志，这部分日志涉及传送到手机端时的情况。SMSC 返回了自身的唯一标识符，这会与请求被确认的时间戳一同被记录下来。根据日志，我们了解到系统在前端节点接收请求后只花了几毫秒，所有消息检查工作就已经顺利通过，然后请求就沿着服务节点送达了 SMSC。

然后，我们根据 SMSC 的唯一标识符检索了从 SMSC 发送到我们系统的传送报告日志。其中反映出了——在那之后的几个小时内，每隔三十分钟，我们记录了一条终端离线（detached）消息，最后是一条短信投递成功报告。根据投诉，短信正是在比赛结束几个小时后收到的。

为什么我们的系统明明处理得非常及时但短信却没能按时投递？答案就在终端离线消息中，这说明用户当时位于信号范围外或关闭了手机。我们花了几分钟的时间来回应这个投诉，证明了我的无辜。要是没有精确的时间戳、唯一的标识符和详细的信息，以及将不同的日志结合分析的能力，那么我们将无法证明自身的清白。请记住这一点，否则当你需要搞明白为何请求会消失在系统中时，由于缺乏线索，要想证明自己清白只能靠运气了。

指标

指标 (metric) 是一组数值数据，是以固定间隔采集并按时间顺序排序的。指标是从你的应用栈中的各个层面采集而来的。你需要从操作系统和网络层采集、中间件层（这包括 Erlang 虚拟机和其他本书中介绍到的库）采集，还需要从业务逻辑层采集。指标可以用在业务中的许多地方，之所以需要它，原因有很多，与你为什么需要警报和日志类似：

- 开发人员使用指标可提高系统的性能和可靠性，并在发生问题后进行故障排除。
- DevOps 工程师用指标来监控系统以检测异常行为并防止故障。
- 运维人员使用指标来预测趋势和使用率峰值，优化硬件成本。
- 营销人员使用它们来研究长期的用户趋势和用户体验。

为了能把指标可视化，想象有这样一个递增的计数器，我们叫它 *login* 好了，每当有人想要登录到系统时，它的值就会增加。如果登录成功，则 *login_success* 计数器也增加，反之若失败，则增加 *login_failure* 计数器。我们还可以更进一步，针对各种不同的登录失败类型设置计数器，如 *bad_password*、*unknown_user*、*user_suspended*、*userdb_error* 等。这些指标有助于我们识别入侵系统的企图，监视欺诈，又或者仅仅是意识到用户体验有些糟糕。如果你发现有大量的未知用户尝试登录，以及出现大量的密码错误，而它们的来源相同，那你可能就想问问负责安全的人员，让他们审查一下日志看看到底出了什么问题。市场部的人可能会想知道，有多少用户在登录时遇到了失败然后重试后最终成功了。其实直接检查日志也能获得这些信息，但是指标为我们提供了直观的提示，让我们意识到问题的存在。

◀ 434

运维团队可能会希望系统负载不要超出可用资源量，希望能够了解 Erlang VM 内存使用量方面的指标。你能够轮询的不仅仅是总的内存用量，还包括其中有多少内存是被进程、系统、原子表、ETS 表、二进制值以及代码所占用的。你甚至能够更进一步区分出，分配用来存储进程和原子的内存里，到底实际使用了多少。

作为开发人员，对于这些问题你可能不会考虑太多，毕竟当务之急是能把开发的系统交付。但是一旦系统上线，而某个并不了解 Erlang 的人却指出为什么此刻可用内存变得很

低时，想象一下，如果系统强大到能让他把内存尖峰、垃圾回收消耗很大的一部分时间，以及用户操作（例如登录）三者联系到一起，将会是多么强大。

图 16-2 展示了 Erlang VM 中不同类型内存的用量，以及总的内存消耗。我们可以清楚地看到，50% 的内存增长是由于进程内存的增长导致的，这可能与系统使用量上升或进程未正确终止产生累积有关。

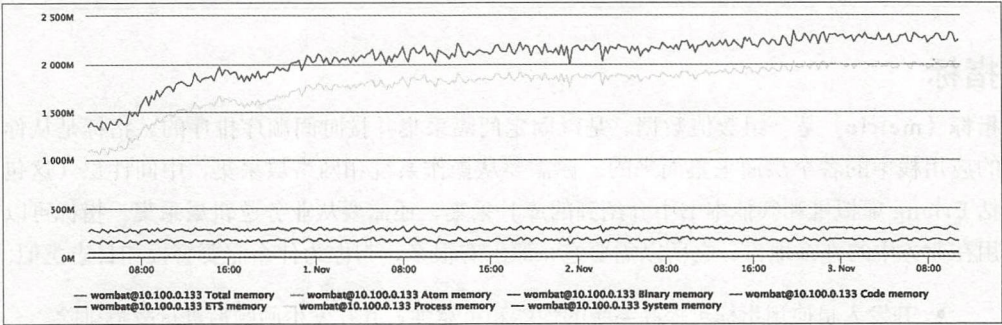


图 16-2: 内存使用

根据用途不同，指标收集哪些值，以及以何种格式收集会有所不同。一种比较典型的值是总量（amount），它是一种离散或连续的可增加也可降低的值。总量的一种常见形式就是计数器（counters），就如我们之前所见过的那样。

标号（gauges）则是一种特殊形式的计数器，它会在特定时间点提供值。对于运维人员来说，我们提供的——系统启动后尝试登录次数——这一信息用处不大，但是我们提供的另一个信息——当前会话数——则很有用。标号的其他典型例子还包括，内存或磁盘的使用量。

435 耗时（time）则是另一种常见的指标，主要用于测量栈中各级的延迟。数据收集器倾向于将时间读数分组成直方图，其包含了（不一定只与时间相关的）值的集合，并且应用了某种形式的统计分析。柱状图里可以显示平均值、最小值和最大值，或者百分比。举一个例子，最快的前 1% 的请求的延迟是怎么样的？最慢的 1% 呢？

第三种类型的指标是特定单位时间内的值。它们通常被称为计量表（meter），提供了一个只能增加的（increment-only）计数器，其值基于平均率（mean rates）与指数加权移动平均数（exponentially weighted moving averages）做了均衡化。这些调整意在确保你不会看到可能出现的尖峰和波谷。

螺旋（spiral）是一种自带滑动窗口计数器的量器，它展示的是某个时间窗口内的计数器增量。如果你想展示最近一分钟内的相关值，则滑动计数器只会读取最近一分钟内的数

据，而忽略早于此时间的数据，并且每秒都会更新。你可以借此展示的值包括：每秒比特率、每秒操作数，以及初始化的会话数等。

每个指标都有与其相关的时间戳。它们被以固定的间隔存入时序数据库 (time series database)，或者从中取出。时序数据库是一种专门针对以时间戳作为索引的数据的优化过的数据库，它让你可以按时间先后顺序访问数据。指标通常随着时间推移不断被汇总和合并，以反映系统级别的概况。你可能会希望收集某类节点上全部的计数器值，这样就能看到过去 15 分钟、1 小时、1 天或者 1 个月内的总请求量。

看看图 16-3 中所示的计数器，它展示的是过去 12 小时内进程消息队列的总长度。该图是为了调查凌晨 3 点 34 分的一次节点崩溃故障而收集数据绘制而成的。节点崩溃的原因是内存耗尽。这些指标不仅允许我们识别出导致崩溃的原因，而且提供了一种运维上的洞察力：如果曾有人监视着进程消息队列长度的变化，则将有 3 小时的时间窗口可以用来察觉和定位这一问题。

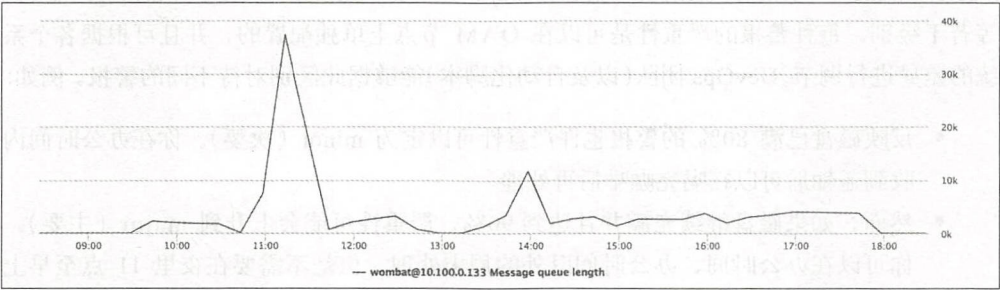


图 16-3：消息队列长度

在 Erlang 中，ETS 表提供了一个原子级操作 `ets:update_counter`，用它可以增加或者减少计数器的值。频繁而快速地调用它们一般没问题，但是要小心在多核架构上其中涉及全局锁问题，这可能会成为瓶颈。指标领域值得推荐的开源应用包括 *folsom* 和 *exometer* 等。它们提供了针对 Erlang VM 的一些基础的系统级指标，而且也允许针对不同节点创建不同的指标。

436



exometer

exometer 由一组应用程序组成，它提供了一个包用于在各个节点中收集指标并导出。它提供了一些可采样的预定义数据点和指标类型，以及用于添加用户自定义指标类型的 API 和回调函数。当想在采样间隔里存储状态时，可以实现一些 *probe*（探针），它们运行在自己的进程空间中，能够存储状态并使用它们来收集指标。指标和数据点可以导出给第三方工具和 API，包括 Graphite、OpenTSDB（通过 Telnet 方式）、AMQP、StatsD 和 SNMP 等。你还可以选择

开发和添加自己的自定义报告器。*exometer* 对内存、CPU 和网络的影响非常小，最小化了对其测量和监控的应用程序的影响。

警报

警报 (alarm) 是与状态相关的事件的一个子集。事件告诉你的是“某事已发生”，而警报则向你指明“某事正在发生”。举一个例子，通过事件你可以发现某个套接字非正常关闭了；但通过警报你可以发现“无法创建到某个外部数据库的套接字连接”这样的事。

当你正在监视的问题显现时，警报就触发了。这包括，举例来说，比如和数据库的套接字连接断开了，或者是启动时无法创建到数据库的连接之类的。警报会一直持续，直到问题解决——也许是问题自己消失了，或者是自动化 / 人工的干涉起了作用——状态又恢复了正常，比如到数据库的网络联通了。当发生这样的状况，我们就说警报被清除了。

与警报相关的一个概念是严重性 (severity)。严重性可以划分为 *cleared* (已清除)、*indeterminate* (不确定)、*critical* (紧急)、*major* (主要)、*minor* (次要)、*warning* (警告) 等若干级别。每种警报的严重性是可以 OAM 节点上单独配置的，并且可根据各个系统的差异进行调节，DevOps 团队 (以及自动化脚本) 能够据此区别对待不同的警报。例如：

- 反映磁盘已满 80% 的警报也许严重性可以定为 *minor* (次要)。你在办公时间内收到通知后可以在喝完咖啡后再处理。
- 然而，如果磁盘继续充满并且达到 90%，严重性可能会上升到 *major* (主要)。你可以在办公时间、办公时间以外的周末处理，但是不需要在夜里 11 点至早上 7 点间处理。
- 如果磁盘已满 95%，则严重性可能会变为 *critical* (紧急)；无论何时发生这种情况，立刻呼叫相关人员，让他们从床上起来进行排查，在节点耗尽空间崩溃之前查明原因并解决。

在另一些系统中，节点崩溃不会影响到可用性和可靠性，一个节点由于磁盘已满导致的崩溃可以等到办公时间再处理——假设有足够的冗余能够保证系统继续承载负载的话。没有通用方案，每个系统的情况都不同，必须区别对待。

警报既可以来自受影响的节点，也可以来自 OAM 节点自身。警报可以是基于阈值的，也可以是基于状态改变的，或者是二者的混合。

基于阈值的警报，内部监视者监视一些指标，当其中一些指标超过限制后，警报就触发了。比如磁盘可用空间不足这种警报，根据超出的值的不同，你可以根据系统的实际需求配置不同的严重级别。其他基于阈值的警报还包括比如内存、套接字数量、端口、打开的文件数、ETS 表数量等。如果你正在监视系统中流经的请求数量，这显然应该是一个不

断增加的计数器，奇怪的是它在过去一分钟内竟然停住了，没有增加，这种状况显示很可能出了什么问题，你自然会想让人去检查一下。

大部分基于阈值的警报都可以在 OAM 节点上进行管理，但也有例外，与你传输数据的频率高低，以及传输的数据量大小有关。举一个例子，进程的消息队列非常长就往往是故障发生的征兆。要监视它们很容易，但是实施监视以及触发警报必须是发生在受监视的节点本身上，因为如果要把所有的消息队列长度都发往 OAM 节点是不可行的。另外还有一些情况下你会希望立刻检测出故障并立刻触发警报，而不必等待 OAM 节点收到指标数据再根据阈值触发警报。

基于状态的警报会在可能有害的状态改变发生时被触发。这包括一类能够指出硬件问题的警报，比如机柜门被打开，或者风扇出故障。其他例子包括与外部 API 的网络连通性，或者数据库无响应，或者节点下线等。

到底需要发送多少警报，发送什么级别的警报，有多少细节包含在这些警报里，这些都由你决定。你可能想要对系统进行正常性检查 (sanity checks)。要是 `.beam`、`.boot`、`.app` 或者 `sys.config` 文件损坏怎么办？除非节点重启，否则你不会发现。虽然说节点重启只需几秒，对于你的系统的上线时间 (uptime) 影响轻微，但是辨别损坏的文件就足以让你和五个九说再见了。隔离出损坏文件并修复可不容易，而且会花很多时间，大大增加你的停机时间。你可能认为自己不会碰到这种情况，先别说大话，因为这种事许多人都碰到过，包括我们！

◀ 438

SASL 包含了一套基本的警报处理器，我们在第 7 章的“SASL 警报处理器”一节已经讨论过。它为你提供了触发和解除警报的功能，但是不包含严重性和相关性方面的功能。SASL 警报处理器背后的思想是——在受影响的节点上管理警报。它们可以被转给 OAM 节点上更复杂的警报应用处理，或者交给外部工具处理。如果你真的选择自己编写 OAM 节点，有一些复杂的细节和可配置功能需要做好，比如处理重复警报、区分严重程度，以及支持交互式操作等。



elarm

elarm application (<https://github.com/esl/elarm>) 是 Erlang 系统里事实上的警报管理器，用于在生产系统中管理警报。它允许你配置严重性与动作，还可以实现通过电子邮件、短信或外部系统（如 Nagios 或传呼机）转发请求的处理程序。用户（或系统）可以确认和清除警报，并且还能添加注释，这有助于 DevOps 团队其他成员的工作。你可以配置警报，将严重性、可能原因、修复动作等信息提供给用户。所有警报事件都会被记录，并且可以查询和过滤当前警报状态。虽然 SASL 警报处理程序对于基础场景也是理想的，但是你应该在 OAM 节点中运行 *elarm*，它是所有警报收集、聚合及合并的聚焦点。这些信息有助于自动或者手动决定应当采取什么行动或是否应提升应对的级别。

没有任何两个系统是相似的。根据功能、流量负载、用户行为模式不同，它们各自需要被以不同的方式进行管理和配置。对于一个系统来说是关键的警报，对于另一个系统来说可能就只能算警告或者根本算不上问题。一旦系统上线，你就需要对警报进行配置和调优。这是很常见的需求，因为你会碰到误报和漏报的情况。所谓误报，就是警报虽然触发了，但是实际上并没有什么问题。出现这种状况的原因可能是由于阈值设置得过度敏感，或者是因为患了妄想症的管理层要求你监视的东西过多了。比如在磁盘被缓慢填充的情况下，一个 70% 磁盘已填满的警报可能会持续几个月，但是实际上根本不需要处理。但是对于磁盘填充速度很快的系统，这样的警报反而可能已经重要到有必要在午夜唤醒相关人员进行处理，否则节点将会在一小时内崩溃。调优系统，消除误报是很重要的，否则许多人就会渐渐习惯它们然后忽略了真正要紧的警报。

439 反之亦然，管理漏报也很重要。所谓漏报，指的就是本应该触发警报，但是却没有。这可能是因为系统中某些部分的阈值配置有问题，或者遗漏了对这些部分的监视。每次出现故障或者服务质量下降后，应考虑对于此类问题该触发何种警报，以及应当监视哪些事件才能提前察觉。

就我们所见的系统，大多数缺乏警报，即使有，也往往是事后添加的。大多数是基于阈值的，少有的基于状态的警报也是依赖于外部探测器（external probes）发送请求到当前系统。在检测和定位异常方面，警报可以说扮演了极其关键的角色，电信领域采取这一做法已有几十年的历史。现在是时候让其他垂直行业吸纳这方面的实践经验了。这将极大促进和支持自动化（下一节会谈到），并且是让系统尝试达到五个九级别可用性时的几大支柱之一。

抢救性支持

支持的自动化实质上是构建一套知识库，通过对外部刺激做出反应使得问题在恶化之前就能得到解决，进而减少服务中断。如果你设计的系统每年允许的下线时间只有几分钟，那么当在设计该系统时，就需要把下线时间作为一个因素纳入考虑范围。当检测到出现问题后，期待着某个人类出现然后手工运行一段脚本来解决问题的思路是不好的。在每年只允许数分钟的下线时间的情况下，运行这种修复脚本的工作只能完全仰仗自动化。自动化的实现，要依靠对指标、事件、警报数据的收集和分析，以及一些配置数据。如果从指标和事件序列中检测出了特定的模式，对应的一系列预定义的动作就会被执行，在问题真正出现前尝试进行抢救。简单的操作包括删除文件、配置负载均衡器，或者部署新的节点以增加吞吐量并降低延迟。

当设计你的 Erlang 系统时，你需要记住要对三个主要的方面进行支持自动化：

- 主动支持自动化 (proactive support automation) 聚焦于减少停机时间, 其主要基于端到端的健康检查和诊断程序。可以通过从外部系统发送请求测试可用性、延迟和功能来实现。
- 抢救性支持自动化 (preemptive support automation) 通过收集特定应用程序的指标、事件、警报和日志并分析数据, 基于分析结果做到在服务中断之前提前预测。一个例子是注意到内存使用量的增加, 这预示着在不久的将来系统可能会耗尽内存, 除非采取适当的纠正措施。这些行动可能包括启用负载调节和背压、请求限流、启动或停止节点以及使用基于容量的部署迁移服务等。
- 自支持自动化 (self-support automation) 指的是可用于进行诊断和故障排除并解决问题的工具和库。主动支持自动化和抢救性支持自动化会调用它们。

主动支持自动化的一个例子是, 通过外部探针模拟用户发送 HTTP 请求, 通过向系统的不同部分发送请求来监视系统是否良好。在我们电子商务的例子中, 探测行为包括测试产品数据库是否能够返回搜索结果, 用户是否能够登录并启动会话, 以及结账及付款过程是否能够成功。毕竟, 世间哪有那种顾客只能浏览商品, 却无法购买的商店呢!

你肯定想在客户意识到出现问题之前先一步发现, 并在社交媒体上出现抱怨的话语之前就开始着手解决问题。请确保探测过程在你的网络之外运行。我们曾经碰到过, 自己在防火墙内检测没有异常, 但是防火墙外的客户却无法访问的情况, 最后找到的原因是交换机有缺陷。

对于抢救性支持, 当警报触发, 或者特定的指标达到阈值时, 如果你明确知道该采取什么措施应对, 那么应该把这个过程自动化。在磁盘填满的例子里, 一旦收到磁盘 80% 已满的警报, 你可以启动日志压缩过程。如果压缩了日志还是不行, 并且磁盘消耗量进一步提升到了 90%, 你可以修改日志的环绕时间 (wraparound time), 并关闭那些不影响服务的部分。如果你不幸碰到了磁盘 100% 填满的警报, 那就开始删除一些不需要并且不会影响系统功能正常的东西。

自动化抢救性支持的其他例子包括, 当现存系统的容量不足时自动部署新的节点, 重新配置负载均衡器, 修改触发负载调节和背压的阈值等。举一个例子, 客户端为了能快速发送日志条目, 使用了 *lager* 库来异步发送, 但是当 *lager* 的邮箱积累到一定量后, 就会把原本的异步调用替换成同步调用, 目的是能够降低生产者的速率, 让 *lager* 能够撑住。

抢救性支持并非必须是完全自动化的。不要低估让你的 DevOps 团队分析日志、警报、指标的价值, 特别是在高峰和超大负载的情况下, 这样做可以预测和避免一些你从未想到过的中断。

干草堆中的针

在客户站点上，我们的节点已经反复崩溃并重启了几个月都没人注意到。原因是我们重构了一些代码，但却没有做浸泡测试，导致漏掉了处理某个端口上的 EXIT 信号——而这个端口同时也是我们用来解析入站 XML 的。默认情况下，Yaws 会回收进程，因此每次进程最终都会收到来自先前请求的数千个 EXIT 消息，因此每次接收到新请求时都必须遍历这些消息。节点经常耗尽内存并崩溃。当重新启动时，邮箱被清除，并且开始累积下一次崩溃。

客户抱怨说，系统有时很慢。我们指责是由于他们的 Windows 服务器导致的速度问题。我们通过外部探针（probe）测试，偶尔会看到系统可用性从 100% 降到 99.999%。我们很少遇到这样的问题——外部探测器每分钟发送一个请求，节点花了几百毫秒来处理它，然后节点花了 3 秒才能重启，所以我们责怪运维团队配错了防火墙导致可用性下降。

即使配备了三重冗余，系统也依然出故障了，只不过我们没有注意到。只有当碰巧登录到其中一个前端节点，并意识到它运行消耗了 100% 的 CPU 占用率而每秒竟只能处理 10 个请求时，我们才发觉碰到了一问题并开始调查。

假如当初对消息队列实施了监控，我们就能立即发现此问题，并阻止其恶化。假如运维人员能看到 CPU 利用率和请求延迟在一段时间内的图像变化，他们也能够留意到问题的出现。假如有人看过日志，他们会看到节点反复崩溃并定期重启。拥有这些信息的价值在于，即使不能立即修复这一 EXIT 信号问题，但至少我们可以通过重新配置 Yaws 来限制其进程回收以临时应对此问题。

这是一个教训，于是在解决了这个问题之后，我们开始监控请求的延迟。这种做法获得了回报，因为我们注意到了，每个小时——恰好每一个小时，延迟从几百毫秒飙升到几秒！经过调查，线索指向了日志文件轮转时的某些同步调用。当执行将文件刷新到磁盘操作时，所有其他请求的处理都被卡住了，因为调用日志进程是同步性质的。我们最终分裂了一个进程来打开新文件并处理所有新日志条目，这样一来我们就可以在后台执行刷新旧文件的操作。

总结

对系统进行监视并不是一件乏味的事情。如果你想要五个九级别的可用性，不要假设有什么是必然没问题的；监视一切，周期性地花时间查阅警报、日志和指标。查阅不仅是手工的，也可以是自动的。你永远不会知道你的工具以及你自己能发现些什么。唯一可

以肯定的是，存在的问题必然会显现出来，而且往往是在你最不希望看到它们的时候显现。

虽然你在函数、进程、应用和节点级别上实现了故障隔离，但是这不等于说进程崩溃是可接受的。“任其崩溃”的方式为你提供了一种编程模型，该模型既简单，同时又能减少崩溃的发生。要确保出现故障时你能及时察觉，并尽快修复。你需要在问题发生前就有所察觉，这样你才有足够的时间在用户发现前解决。

最后，不要把时间浪费在大海捞针上。有这些全部的数据在手，当异常问题出现时你也就能够证明自己的无辜了。

就这么简单！是谁说设计可伸缩、高可用的系统很难？你所需要做的只是下面这简单的10步：

1. 将系统功能分割为一系列可管理的、独立的节点。
2. 选择一种分布式架构模式。
3. 为你的节点间、节点家族间以及集群间的相互通信选择适当的网络协议。
4. 定义清楚节点的接口、状态和数据模型。
5. 针对你的每一个节点的每一个接口函数，挑选一种重试策略。
6. 针对全部数据和状态，挑选在不同节点家族、不同集群、不同节点类型间共享它们时的合适策略，特别是要把重试策略纳入考量。
7. 设计你的系统蓝图，留意系统规模扩展/收缩时各类节点的比例关系。
8. 分辨出哪些地方需要应用背压和负载调节。
9. 定义你的 OAM 方式，定义系统方面和业务方面的警报、日志和指标。
10. 明确在哪些地方使用自动化支持。

最后，当所有这一切都已做到位并运作良好时，记得随着需求的不断演变要重新审视当初你所做出的权衡和假设。当需要时，在其中添加更多的弹性和可见性。识别出每一次中断的原因，并把它们纳入你的监视系统，以在早期获得警告，同时增强你的软件和基础设施的可恢复性，确保同样的中断不会再次发生。

要做更进一步的阅读，我们建议你看看 *lager*、*elarm*、*exometer*、*folsom* 的文档。这些文档可以在这几个项目在 GitHub 上的仓库里找到。*Stuff Goes Bad, Erlang in Anger* 是 Fred Hébert 写的一本在线电子书 (<https://www.erlang-in-anger.com/>)，我们强烈推荐你阅读它。Erlang/OTP 系统文档中有 OAM 原则方面的用户指南，主要聚焦于 SNMP。这份指南得配合其他一些运维应用程序的文档一起阅读，包括 *os_mon*、*otp_mibs*、*snmp* 等。

接下来是什么

这是我们计划中编写的最后一章——至少目前来说是这样。把你从本书中学到的知识用到实践中去，基于 Erlang/OTP 和其编程模型设计出你的可伸缩、高可用的系统。当这一刻到来的时候，你也就相当于成为本书续章的撰写者。这么做的时候，记住 Håkan Millroth 曾在早期的一次 Erlang 用户会议上说过的话：项目的成功源自优良工具、同道中人和一点点智慧。你已经了解了这么多优秀的工具，而你本身就很有智慧，可能也有一些好的志同道合者。我们期待着听到你的成功故事！感谢你阅读到这里，祝你好运！

Symbols

+ character, 290
 ++ list operator, 28
 -- list operator, 28
 := map update operator, 42
 => map insert/update operator, 42
 ? (question mark), 42

A

abcast() function, 98
 abnormal process termination
 about, 36, 54
 cyclic restarts after, 173
 handling errors and invalid return values, 158
 supervisors and, 199
 ACID acronym, 396
 add_handler() function, 150
 add_patha() function, 216
 add_sup_handler() function, 159, 163
 administration state, 227
 alarm handlers, 165-166, 438
 alarming
 about, 18, 203
 monitoring via, 426, 436-439
 alarm_handler module, 165, 167
 Allen, Mark, 371
 Alpern, Bowen, 399
 Amdahl's law, 406
 americano coffee, 131
 amount (metrics), 434
 anonymous functions, 25-26
 app files
 about, 206, 207, 314
 application versions and, 272
 appup files and, 333
 base station controller, 215
 ebin directory and, 208
 properties supported, 213-216
 setting environment variables, 220
 application controller, 205, 279
 -Application flag, 292, 350
 application master, 205
 application module
 about, 16, 209
 starting applications, 210, 216-217
 stopping applications, 211-213
 application resource files (see app files)
 application:ensure_all_started() function, 216
 application:get_all_env() function, 219
 application:get_application() function, 219
 application:get_env() function, 214, 219
 application:load() function, 210, 280
 application:set_env() function, 221
 application:start() function, 165, 210, 221, 226
 application:start_boot() function, 280
 application:stop() function, 211, 222
 application:which_applications() function, 212, 213
 applications
 about, 204-205
 callback module, 209-213
 combining with supervisors, 230
 distributed, 222-226
 environment variables, 219-221
 Erlang nodes and, 11
 included, 228
 loading, 212

†: 索引所列页码为本书英文版页码, 请参照正文侧边用“□”表示的原书页码。

- resource files, 213-216
- SASL, 231-239
- sources for, 203
- standard releases and, 270
- start phases, 226-228
- starting, 205, 210, 216-217
- stopping, 211-213
- structural overview, 206-209
- termination strategies, 221
- tools supporting Erlang, 7-9
- types of, 205
- upgrading, 352
- version considerations, 272
- apply() function, 280
- apply_after() function, 97
- apply_interval() function, 97
- appmon (application monitor), 218
- appup files
 - about, 208, 314, 327, 334-337
 - high-level instructions, 337-339
 - low-level instructions, 342-343
- args_file flag, 292
- arguments, passing to runtime system, 290-296
- Armstrong, Joe, 7, 117
- ASN.1 program, 8
- asynchronous events
 - coffee machine example, 122-125
 - sending, 131, 153-155
 - sending to all states, 136
- asynchronous message passing, 12, 31, 84
- async_shell_start flag, 292
- at least once approach, 389, 400
- at most once approach, 389, 401
- atomicity (ACID), 396
- authentication, 361, 378
- availability (see high availability)

B

- back-end nodes, 361, 362, 378
- back-off algorithms, 386
- backpressure, load regulation and, 419-421
- Bailis, Peter, 400
- balancing systems, 409, 414-416
- base station controller example, 215, 270
- Basho website, 371
- Basic Operating System (BOS), 4
- basic target systems, 264
- +Bc emulator flag, 290
- +Bd emulator flag, 291

- beam files, 207, 214, 314
- BEAM virtual machine
 - about, 7
 - balancing systems, 414
 - garbage collection and, 109
 - schedulers and, 32
- behavior directive
 - about, 79
 - applications and, 210, 230
 - event handlers and, 150
 - FSMs and, 127
 - spellings recognized, 79
 - supervisors and, 176, 230
- behavior module, 57-60
- behaviors
 - about, 10
 - creating, 256
 - design patterns and, 53, 56-60
 - extracting generic, 60-69
 - generic FSM example, 127-140
 - generic server, 69-72
 - globally registered, 97
 - hibernating, 97
 - implementing, 255-260
 - linking, 99
 - message passing, 72-75
 - monitoring, 114
 - processes and, 10, 53-56
 - spawn options, 108-114
 - sys module support, 101-108
 - TCP stream example, 256-260
 - timeouts, 114
- +Bi emulator flag, 291
- bidirectional links, 35
- BIFs (built-in functions), 9, 32
- bin directory, 266, 289
- BINDIR environment variable, 268
- boot file
 - about, 314
 - alternative, 282
 - creating, 274-277
 - creating releases, 273
 - init module, 302
 - make_script parameters, 281-283
 - script files, 277-281
- boot flag, 292
- BOS (Basic Operating System), 4
- bottlenecks
 - data volume and, 373

- distributed Erlang and, 366, 373
- finding, 416-417
- message queues and, 417
- remedies for, 363
- stress testing, 412
- Brewer, Eric, 398
- British Telecom, 383
- built-in functions (BIFs), 9, 32

C

- c:erlangrc() function, 280
- call timeouts, 91-95
- call() function
 - gen_event module, 156
 - gen_server module, 78, 82-83, 91-94
- callback functions
 - about, 77
 - behavior example, 127-140
 - defining states, 132-135
 - event handlers and, 149
- callback modules
 - about, 57-60, 77
 - applications, 209-213
 - directives in, 79, 127
 - event handlers and, 149
 - FSMs and, 126
- CAP theorem, 398
- capacity planning
 - about, 409-411
 - balancing systems and, 414-416
 - capacity testing and, 412-413
 - finding bottlenecks, 416-418
 - system blueprints and, 419
- capacity testing, 412-413
- case expression, 23
- cast() function, 78, 84
- catch-all clauses, 33-34
- change_code() function, 348
- chaos monkey tool, 199
- check_childspecs() function, 186
- check_install_release() function, 348
- chess board list comprehension, 28
- child processes
 - about, 10
 - dynamic, 186-193
 - starting, 171
 - trapping exits, 171
- child specification (supervisors)
 - dynamically creating, 186-193

- elements of, 171, 183-186
- client API, 64-66
- cloud computing environments, 364
- cluster blueprints, 419
- code:add_patha() function, 216
- code:lib_dir() function, 207
- code:load_file() function, 44, 293, 318
- code:purge() function, 45, 318
- code:root_dir() function, 265
- code:soft_purge() function, 45
- code_change() callback function, 79
- coffee machine FSM example
 - adding states, 323-326
 - creating release upgrade, 326-350
 - software upgrades, 320-323
 - states described, 119-125
 - stepping through behaviors, 127-140
 - supervisor example, 172
- common_test application, 9
- compile:file() function, 44
- config flag, 292
- configuration files
 - about, 314
 - distributed applications and, 222
 - environment variables and, 219
- connect_all flag, 292
- consensus protocols, 396
- consistency
 - forms of, 396
 - in ACID acronym, 396
 - in CAP theorem, 398
 - tradeoffs with availability, 400-401
- consistent hashing system, 368
- counters (metrics), 434
- count_children() function, 185
- CPU-bound nodes, 415
- crash reports (SASL), 237
- create_RELEASES() function, 347
- CTRL-d, 289
- CXC product-numbering scheme, 212
- cyclic restarts, 173

D

- Däcker, Bjarne, 5
- data networks, 364
- data sharing, 392-398
- dbg debugger, 9, 270
- deadlocks, 94-95
- debugger tool, 8, 270

- debug_options() function, 245, 249
 - delete() function, 156
 - delete_child() function, 189, 190, 337
 - delete_handler() function, 152
 - demilitarized zone (DMZ), 364
 - demonitor() function, 37, 75
 - detached flag, 292
 - DETS tables, 46
 - Deutsch, Peter, 365
 - dialyzer tool, 8, 79, 267
 - diameter stack, 8
 - directory structure
 - applications, 206-209
 - releases, 265-269
 - disaster recovery guidelines, 387
 - distributed application controller, 222-226
 - distributed applications, 222-226
 - distributed environments
 - about, 359
 - fallacies of distributed computing, 365
 - interfaces and, 377-380
 - networking in, 363-377
 - node types and families, 360-363
 - upgrading in, 351
 - distributed Erlang
 - about, 366
 - Riak Core and, 367-371
 - scalable, 371
 - distribution
 - about, 2, 12-13
 - directory structure, 207
 - Erlang nodes and, 48-50
 - DMZ (demilitarized zone), 364
 - dotted version vectors (DVVs), 370
 - DTrace probe, 9
 - duplicate requests, 390
 - durability (ACID), 396
 - DVVs (dotted version vectors), 370
 - dynamic children, 186
 - dynamic clusters, 366
 - dynamic modules, 255
- ## E
- +e emulator flag, 291
 - ebin directory
 - about, 206
 - application structure and, 206
 - beam files and, 208, 214
 - releases and, 266
 - starting applications and, 216
 - echo:go() function, 30
 - echo:loop() function, 30
 - ei program, 8
 - elarm application, 9, 203, 438
 - elasticity, 407
 - eldap application, 8
 - element() function, 38
 - emacs editor, 9
 - embedded mode, 17, 279, 293
 - embedded target systems, 283, 293
 - emulator flags, passing to runtime system, 290-296
 - emulator, upgrading, 352
 - emu_args flag, 292
 - enif_schedule_nif() C API function, 409
 - ensure_all_started() function, 216
 - env flag, 293, 298
 - environment variables
 - applications and, 219-221
 - distributed applications and, 222
 - heart and, 298
 - release handling and, 268
 - upgrading, 350
 - epmd daemon, 267
 - epp (Erlang preprocessor), 42
 - Ericsson Computer Science Laboratory, 1, 4
 - erl command
 - s flag, 264
 - about, 22, 263, 266
 - bin directory and, 289
 - erl file extension, 314
 - Erlang loader, 300-302
 - Erlang nodes
 - about, 7, 11, 203
 - connections and visibility, 49-50
 - distributed environments and, 360-363
 - distribution and, 48-50
 - naming and communication, 48
 - startup failures and, 205
 - Erlang preprocessor (epp), 42
 - Erlang programming language
 - about, 6
 - catch-all clauses, 33-34
 - defining the problem, 2-4
 - distribution, 12-13, 48-50
 - ETS tables, 45-48
 - functional influence, 25-28
 - infrastructure, 12

- links and monitors for supervision, 34-38
- macro facility, 42
- maps, 41-42
- message passing, 29-32
- multicore systems, 13, 32
- pattern matching, 21-25
- processes, 29-32
- records, 38-41
- recursion, 21-25
- upgrading modules, 43-45
- Erlang Syntax Tools modules, 8
- Erlang/OTP (see OTP framework)
- Erlang: the Movie, 117
- erlang:apply() function, 280
- erlang:demonitor() function, 37, 75
- erlang:element() function, 38
- erlang:exit() function, 34, 36
- erlang:link() function, 35, 49
- erlang:make_ref() function, 38, 73
- erlang:map_size() function, 42
- erlang:monitor() function, 37, 74
- erlang:monitor_node() function, 50
- erlang:node() function, 50
- erlang:nodes() function, 50
- erlang:process_flag() function, 36, 85
- erlang:register() function, 97
- erlang:self() function, 30, 244
- erlang:send_after() function, 97
- erlang:set_cookie() function, 49
- erlang:spawn() function, 29, 49
- erlang:spawn_link() function, 35, 49
- erlang:spawn_opt() function, 108-114
- erlang:system_info() function, 272, 291
- erlang:system_monitor() function, 373, 417
- erlang:unlink() function, 35
- erlangrc() function, 280
- erlc program, 266
- erlexec binary, 266
- ERL_CRASH_DUMP_SECONDS environment variable, 298
- erl_interface, 8
- erl_prim_loader module, 279, 300
- erl_prim_loader:get_file() function, 279
- error exception, 34
- error reports (SASL), 236
- error_logger module, 279
- erts directory
 - about, 7
 - contents of, 266
 - mapping to, 269
- escript program, 267
- et (event tracer) tool, 8
- etop tool, 416
- ETS (Erlang Term Storage) tables, 45-48, 156, 191
- ets:delete() function, 156
- ets:give_away() function, 191
- ets:match() function, 47
- ets:new() function, 45
- ets:update_counter() function, 436
- eunit tool, 9
- eval flag, 293, 295
- event handlers
 - about, 11, 148
 - adding, 150-152
 - callback modules and, 149
 - deleting, 152
 - event managers and, 149
 - generic, 149-164
 - gen_event module and, 15, 149
 - handling errors and invalid return values, 158-160
 - retrieving data, 156-157
 - SASL alarm handler, 165-166
 - sending synchronous and asynchronous events, 153-155
 - swapping, 160-162
- event managers
 - about, 11, 147
 - event handlers and, 149
 - generic, 149-164
 - gen_event module and, 15, 149
 - handling errors and invalid return values, 158-160
 - SASL alarm handler, 165-166
 - sending synchronous and asynchronous events, 153-155
 - starting and stopping, 149
- events
 - about, 118, 147-149
 - asynchronous, 122-125, 131, 136, 153-155
 - logging, 19
 - sending, 131-139
 - synchronous, 131, 138-139, 153-155
 - system messages, 103
- eventual consistency, 396
- exactly once approach, 389, 400
- exception handling

- authentication, 379
- event handlers and, 158-160
- generic supervisors and, 169
- negative numbers and, 33-34
- unhandled messages, 87
- exit exception, 34
- exit signals, 35, 85, 99, 441
- exit() function, 34, 36
- exometer application, 9, 203, 436
- exponential back-off algorithms, 386
- export directive, 127

F

- factorials, computing for positive numbers, 21
- failovers, 222
- false negatives, 439
- false positives, 438
- fault tolerance
 - about, 3, 383, 401
 - availability and, 384-385
 - distribution and, 12-13
 - supervision trees and, 170
- Fibonacci, 386
- FIFO queue strategy, 422
- file extensions, 314
- file() function, 44
- filelib:wildcard() function, 40
- finite state machines (see FSMs)
- Fischer-Lynch-Paterson (FLP) Impossibility result, 399
- flags, passing to runtime system, 290-296
- FLP (Fischer-Lynch-Paterson) Impossibility result, 399
- folsom application, 9, 203, 436
- format() function, 89, 104
- frequency allocator example
 - differentiating among allocators, 163
 - extracting generic behaviors, 60-69
 - message passing, 83
 - supervision structure, 174, 180-182
 - tracing and logging, 101-102
- front-end nodes, 361, 362, 378
- FSMs (finite state machines), 117
 - (see also coffee machine FSM example)
 - about, 11, 15, 117-119
 - adding states, 323-326
 - behavior example, 127-140
 - defining states, 132-135
 - generic FSMs, 125-126
 - generic servers versus, 119
 - mutex states and, 242
 - phone controllers example, 141-145
 - sending events, 131-139
 - starting, 127-130
 - terminating, 139-140
- full sweep garbage collection, 110
- function_clause runtime error, 87

G

- garbage collection, 33, 109-113
- gauges (metrics), 434
- generic behaviors
 - about, 60-62
 - client functions, 64-66
 - functions internal to server, 68
 - server loop, 66-68
 - starting the server, 62-64
- generic event handlers
 - about, 149
 - adding, 150-152
 - deleting, 152
 - handling errors and invalid return values, 158-160
 - retrieving data, 156-157
 - SASL alarm handler, 165-166
 - sending synchronous and asynchronous events, 153-156
 - swapping, 160-162
- generic event managers
 - about, 149
 - handling errors and invalid return values, 158-160
 - SASL alarm handler, 165-166
 - sending synchronous and asynchronous events, 153-156
 - starting and stopping, 149
- generic FSMs
 - about, 125-126
 - behavior example, 127-140
 - timeouts, 130, 135-136
- generic servers
 - about, 11, 77
 - behavior directives, 78
 - call timeouts, 91-95
 - deadlocks, 94-95
 - FSMs versus, 119
 - globally registered processes, 97
 - linking behaviors, 99

- message passing, 82-89
 - server module example, 69-72
 - starting a server, 80-82
 - terminating, 89-91
 - timeouts, 95-97
 - gen_event module
 - about, 15, 149
 - adding event handlers, 150-152
 - deleting event handlers, 152
 - handling errors and invalid return values, 158-160
 - retrieving data, 156-157
 - sending synchronous and asynchronous events, 153-156
 - starting and stopping event managers, 149
 - swapping event handlers, 160-162
 - gen_event:add_handler() function, 150
 - gen_event:add_sup_handler() function, 159, 163
 - gen_event:call() function, 156
 - gen_event:delete_handler() function, 152
 - gen_event:notify() function, 153
 - gen_event:start() function, 149
 - gen_event:start_link() function, 149
 - gen_event:stop() function, 149, 152
 - gen_event:swap_handler() function, 160
 - gen_event:swap_sup_handler() function, 161
 - gen_event:sync_event() function, 156
 - gen_event:sync_notify() function, 153
 - gen_fsm module
 - about, 15, 242
 - behavior example, 127-140
 - selective receives and, 137
 - sending events, 131-139
 - starting the FSM, 127-130
 - terminating, 139-140
 - gen_fsm:reply() function, 138
 - gen_fsm:send_all_state_event() function, 136
 - gen_fsm:send_event() function, 131
 - gen_fsm:start() function, 129
 - gen_fsm:start_link() function, 127-130
 - gen_fsm:sync_send_all_state_event() function, 138-139
 - gen_fsm:sync_send_event() function, 138-140
 - gen_rpc application, 374
 - gen_server module
 - about, 14, 77
 - behavior directives, 78
 - call timeouts, 91-95
 - deadlocks, 94-95
 - generic server timeouts, 95-97
 - globally registered processes, 97
 - linking behaviors, 99
 - message passing, 82-89
 - starting a server, 80-82
 - system messages, 103
 - terminating generic servers, 89-91
 - gen_server:abcast() function, 98
 - gen_server:call() function, 78, 82-83, 91-94
 - gen_server:cast() function, 78, 84
 - gen_server:multi_call() function, 98
 - gen_server:reply() function, 89, 103
 - gen_server:start() function, 99
 - gen_server:start_link() function, 78, 80-82
 - gen_tcp module, 373
 - get flag (sys module), 102, 108
 - gethostbyaddr() function, 40
 - gethostbyname() function, 40
 - get_all_env() function, 214, 219
 - get_application() function, 219
 - get_argument() function, 296
 - get_arguments() function, 296
 - get_env() function, 219
 - get_file() function, 279
 - get_plain_arguments() function, 296
 - get_state() function, 106, 108
 - get_status() function
 - init module, 279, 302
 - sys module, 105, 108
 - give_away() function, 191
 - global module, 98, 129
 - global name server, 373
 - global:register_name() function, 98, 372
 - global:send() function, 98
 - global:unregister_name() function, 98
 - global:whereis_name() function, 98
 - go() function, 30
 - gossip protocol, 369
 - gproc application, 376
 - Guerra, Mariano, 371
- ## H
- halt() shell command, 224, 291
 - handle_call() callback function
 - gen_event and, 157
 - gen_server and, 78, 82-83, 88, 92
 - handle_cast() callback function, 78, 84
 - handle_debug() function, 250, 251

handle_event() callback function, 153
 handle_info() callback function, 154
 handle_sync_event() callback function, 138, 140
 handle_system_message() function, 249
 handle_system_msg() function, 350
 -heart flag, 293
 heart module
 about, 17, 267, 296-299
 kernel process and, 279
 unhandled messages and, 87
 VM termination and, 182
 HEART_BEAT_TIMEOUT environment variable, 298
 HEART_COMMAND environment variable, 298
 Heriot-Watt University (Scotland), 200
 hibernate() function, 255
 hibernating
 behaviors, 97
 special processes, 255
 -hidden flag, 49, 293
 high availability
 about, 3, 383
 fault tolerance and, 384
 in CAP theorem, 399
 reliability, 387-392
 resilience and, 385
 sharing data and, 392-398
 strategies for, 18
 tradeoffs with consistency, 400-401
 higher-order functions, 25
 hinted handoffs, 369
 histograms, 435
 horizontal scaling, 405-407

I

i() shell command, 416
 I/O-bound nodes, 415
 ic (IDL compiler), 9
 idempotence, 390
 -include directive, 208
 include directory, 206, 208
 included_applications parameter, 228-230
 inet module, 300
 inet:gethostbyaddr() function, 40
 inet:gethostbyname() function, 40
 init module, 279, 302
 init() callback function

about, 78, 79
 gen_fsm example, 128-130
 gen_server example, 80-82
 supervisor example, 176
 supervisor_bridge example, 194
 synchronous nature of, 197

init:get_argument() function, 296
 init:get_arguments() function, 296
 init:get_plain_arguments() function, 296
 init:get_status() function, 279, 302
 init:reboot() function, 302, 348
 init:restart() function, 302, 348
 init:stop() function, 302
 initializing processes, 54
 init_ack() function, 244
 init_debug flag, 290
 -init_debug flag, 293
 init_request() callback function, 257
 install() function, 104, 108, 251
 install_release() function, 338, 342, 347
 interactive mode, 17, 293
 interfaces, distributed environments and, 377-380
 invalid return values, 158-160
 io:format() function, 89, 104
 isolation (ACID), 396

J

jinterface program, 8
 jobs regulation application, 9
 Jobs scheduler, 422

K

kernel application
 about, 7
 alternative boot files, 282
 application resource files and, 214
 environment variables and, 222
 error_logger service, 280
 release resource files and, 269
 kernel process, starting, 279
 kernel_load_completed command (script file), 279

L

lager application, 9, 203, 430
 Lamport, Leslie, 399
 last call optimization, 23

- latency, 410, 413, 414
- LDAP (Lightweight Directory Access Protocol), 8
- leex lexical analyzer generator, 9
- lib directory, 207, 266
- libraries and tools, 7-9
- library applications, 205
- lib_dir() function, 207
- lifecycle of processes, 54
- LIFO queue strategy, 422
- Lightweight Directory Access Protocol (LDAP), 8
- link() function, 35, 49
- linking
 - behaviors, 99
 - processes, 35-37
- list comprehensions, 27-28
- lists
 - generating and testing, 25, 27-28
 - printing elements of, 22
- Little's Law, 420
- load application type, 271
- load regulation and backpressure, 419-421
- load testing, 413
- load() function, 210, 280
- load_file() function, 44, 293, 318
- log directory, 289
- log() function, 102, 108, 250
- logger callback module, 152
- logic nodes, 361, 362, 378
- logrotate tool, 430
- logs and logging
 - application support, 203
 - monitoring via, 426, 428-432
 - of events, 19
 - SASL, 239
 - special processes, 250
- log_to_file() function, 102, 108
- loop() function, 30
- looping behavior, 23
- Lynch, Nancy, 398

M

- m(module) command, 213
- macro facility, 42
- make:files() function, 318
- make_permanent() function, 345, 349
- make_ref() function, 38, 73
- make_relup() function, 340

- make_script() function, 274-277, 281, 300, 339
- make_tar() function, 284, 340, 346
- map collection type, 41-42
- map_size() function, 42
- match() function, 47
- md5 digest, 319
- megaco stack, 8
- memory management, 109-113, 373, 417
- memory-bound nodes, 415
- message passing
 - about, 7
 - asynchronous, 12, 31, 84
 - behaviors and, 72-75
 - generic servers, 82-89
 - other messages, 85
 - processes and, 29-32
 - synchronous, 88
 - synchronous message, 82-83
 - system messages, 249, 373
 - unhandled messages, 86-88
- meters (metrics), 435
- metrics
 - application support, 203
 - importance of, 18
 - monitoring via, 426, 433
 - observer tool and, 218
 - storing, 156-157
- microservices, 375
- Millroth, Håkan, 443
- Mnesia distributed database, 8, 198, 399
- mnesia:transform_table() function, 333
- mode flag, 293
- module directive, 127
- modules, 57
 - (see also callback modules)
 - behavior, 57-60
 - missing or corrupt, 205
 - storing definitions in, 22
 - two-module version limit, 318
 - upgrading, 43-45
- Modules element (child specification), 184
- module_info() callback function, 319
- module_not_found error, 155
- MongooseIM chat server, 203
- monitor() function, 37, 74
- monitoring
 - behaviors, 114
 - observer tool and, 218
 - processes, 37-38



- purpose of, 425-427
 - via alarms, 426, 436-439
 - via logs and logging, 426, 428-432
 - via metrics, 426, 433
- monitor_node() function, 50
- monotonic read model, 396
- monotonic write model, 396
- multicall() function, 374
- multicore systems
 - memory spikes and, 417
 - scaling challenges, 32, 407
 - SMP and, 13
- multi_call() function, 98
- mutable variables, 24
- mutex (mutual exclusion)
 - about, 242-244
 - example, 251-255
 - states supported, 247

N

- Name element (child specification), 183
- name flag, 49, 294, 300
- namespaces, 372
- native implemented functions (NIFs), 408
- negative numbers, 33
- net kernel process, 366, 373
- networking in distributed environments
 - about, 363-365
 - distributed Erlang, 366-372
 - peer-to-peer architectures and, 376
 - service orientation and microservices, 375
 - sockets and SSL, 373-375
- net_kernel module, 49
- new() function, 45
- NIFs (native implemented functions), 408
- Nkcluster application, 371
- Nkdist library, 371
- node attributes, 372
- node families, 362
- node() function, 50
- nodes (see Erlang nodes)
- nodes() function, 50
- none application type, 271
- normal applications, 205
- normal process termination, 36, 54, 199
- nostick flag, 293
- notify() function, 153
- no_dot_erlang.boot file, 282

O

- O&M networks, 364
- OAM (Operation, Administration, and Maintenance), 426
- Object Request Brokers, 9
- observer tool, 8, 218, 416
- observer:start() function, 217, 295
- odbc interface, 8
- one_for_all strategy, 180
- one_for_one strategy, 179
- Operation, Administration, and Maintenance (OAM), 426
- orber broker, 9
- os_mon application, 8, 270
- OTP behaviors (see behaviors)
- OTP framework, 6
 - (see also Erlang programming language)
 - about, 4-6
 - defining the problem, 2-4
 - Erlang nodes (see about)
 - name origin, 5
 - scaling and, 408
 - system design principles, 10-11
 - tools and libraries, 7-9
- otp_mibs, 8

P

- +P emulator flag, 291
- p2p (peer-to-peer) architectures, 376
- pa flag, 294
- partition tolerance (CAP theorem), 399
- parsetools application, 9
- path command (script file), 279
- pattern matching and recursion, 21-25
- Paxos protocol, 396
- peer-to-peer (p2p) architectures, 376
- percept tool, 416
- perimeter networks, 364
- permanent application type, 222, 271
- phone controllers example, 141-145, 187-189
- pids (process identifiers), 29
- plain arguments, 290
- plain_fsm library, 138
- pman (process manager), 218
- point of failure, 387-388, 407
- poolboy library, 363, 374
- preemptive support automation, 425, 439-440
- preleases (release candidates), 272
- preLoaded command (script file), 278



- prep_stop() callback function, 228
- primary-primary replication, 398
- primary-secondary replication, 398
- primLoad command (script file), 279
- print flag, 102, 108
- printing list elements, 22
- priv directory, 206, 208, 272
- proactive support automation, 439
- process heap, 111
- process ID, 367
- process identifiers (pids), 29
- processes
 - application controller and, 205
 - behaviors and, 10, 53-56
 - design patterns and, 53, 56-60
 - events and, 147
 - globally registered, 97
 - initializing, 54
 - lifecycle of, 54
 - links and monitors for supervision, 34-38
 - memory and, 32
 - message passing and, 29-32
 - monitoring, 37-38
 - non-OTP-compliant, 194-195
 - restarting, 173
 - short-lived, 195-197
 - spawning, 29, 54, 108-114, 170
 - special, 16, 195, 241-255, 350
 - terminating, 36, 54, 158-160, 173
- ProcessType element (child specification), 184
- process_flag() function, 36, 85
- proc_lib module
 - adding non-OTP-compliant processes, 195
 - starting special processes, 242, 244-246
 - user-defined behaviors and, 256
- proc_lib:handle_system_msg() function, 350
- proc_lib:hibernate() function, 255
- proc_lib:init_ack() function, 244
- proc_lib:spawn() function, 244
- proc_lib:start() function, 244
- proc_lib:start_link() function, 244
- progress reports (SASL), 236
- PropEr tool, 169
- public_key module, 208
- purge() function, 45, 318
- pz flag, 294

Q

- +Q emulator flag, 291

- q() shell command, 291
- question mark (?), 42
- QuickCheck tool, 169

R

- +R emulator flag, 291
- RabbitMQ message broker, 203
- race conditions, 299
- Raft protocol, 396
- ranch library, 363
- random back-off algorithms, 386
- rb:start() function, 239
- read your own writes consistency level, 396
- rebar3 tool
 - about, 303
 - creating releases, 308-310
 - generating a release project, 304-307
 - releases with project dependencies, 310-312
 - upgrades with, 353-355
- reboot() function, 302, 348
- reboot_old_release() function, 349
- record directive, 39
- records
 - about, 38-41
 - correct versions, 41
 - upgrading, 333
- recursion and pattern matching, 21-25
- reduction count, 32
- register() function, 97
- register_name() function, 98, 372
- regs() shell command, 416
- regular expressions, special characters and, 334
- rel files, 269, 270, 314
- RELDIR environment variable, 269
- release candidates (preleases), 272
- release handling
 - arguments and flags, 290-302
 - configuring on target, 287-289
 - creating release packages, 283-287
 - creating release upgrades, 326-350
 - creating releases, 273
 - init module, 302
 - rebar3 tool and, 303-312
 - release and application versions, 272
 - release directory structure, 265-269
 - release resource files, 269-272
 - software upgrades, 318-326
 - start scripts, 287-289
 - upgrading modules, 43-45



- RELEASE project, 360, 371
- release upgrades
 - appup files, 333-337
 - code to upgrade, 330-333
 - creating, 326-329
 - high-level instructions, 337-339
 - installing, 343-345
 - low-level instructions, 342-343
 - release_handler module, 346-349
 - relup files, 339-342
 - upgrading environment variables, 350
- releases directory, 282, 344, 346
- release_handler module, 346-349
- release_handler:check_install_release() function, 348
- release_handler:create_RELEASES() function, 347
- release_handler:install_release() function, 338, 342, 347
- release_handler:make_permanent() function, 345, 349
- release_handler:reboot_old_release() function, 349
- release_handler:remove_release() function, 345, 349
- release_handler:unpack_release() function, 344, 347
- release_handler:which_releases() function, 349
- reliability, 12, 387-392, 401
- reltool (release management tool), 9, 308, 315
- relup files, 314, 327, 339-342
- relx tool, 308
- remove() function, 105, 108
- remove_release() function, 345, 349
- remsh flag, 294
- replace_state() function, 107, 108
- reply() function
 - gen_fsm module, 138
 - gen_server module, 89, 103
- resilience, 385, 401
- resource blueprints, 419
- resource files
 - applications, 213-216
 - releases, 269-272
- restart specification (supervisors), 179-183
- restart() function, 302, 348
- RestartType element (child specification), 183
- restart_child() function, 190
- restart_new_emulator instruction, 352

- RESTful APIs, 363
- rest_for_one strategy, 177, 180, 337
- resume() function, 106, 108
- return values, invalid, 158-160
- rex (RPC server), 366, 373
- RFC 6733, 8
- Riak Core framework
 - about, 9, 13, 203
 - distributed Erlang and, 367-371
 - scalability of, 411
- ROOT environment variable, 279
- ROOTDIR environment variable, 268
- root_dir() function, 265
- rpc:multicall() function, 374
- rr shell command, 40
- run flag, 295
- runtime_tools application, 9, 207, 270
- run_erl binary, 267

S

- s flag, 294, 295
- Safetyvalve framework, 9, 422
- sasl application
 - about, 7, 15, 212
 - alarm handler, 165-166, 438
 - alternative boot files, 282
 - crash reports, 237
 - error reports, 236
 - progress reports, 236
 - release resource files and, 269
 - services supported, 231-235
 - supervisor reports, 238
- sasl application resource file, 213-215
- SASL logs, 239
- scalability
 - about, 2, 405
 - capacity planning and, 409-419
 - distributed Erlang and, 366
 - distribution and, 13
 - horizontal and vertical, 405-407
 - load regulation and backpressure, 419-421
 - Riak Core and, 370
 - short-lived processes and, 195-197
 - strategies for, 18
- scaling out, 407
- scaling up, 407
- schedulers, 32, 422
- Schneider, Fred B., 399
- script files, 277-283, 314



- script2boot() function, 282
- SD (Scalable Distributed) Erlang framework
 - about, 9, 13, 371
 - scalability of, 411
- secret cookies, 49
- selective receives, 137
- self() function, 30, 244
- self-support automation, 440
- semantic node types, 360
- semi-explicit placement, 372
- send() function, 98
- send_after() function, 97
- send_all_state_event() function, 136
- send_event() function, 131
- send_interval() function, 97
- server module example, 69-72
- service metadata, 375
- service nodes, 361, 363, 378
- service-level agreements, 420
- service-oriented architecture (SOA), 12, 375
- session ID, 362
- setcookie flag, 49, 294
- set_cookie() function, 49
- set_env() function, 221
- share-everything architecture, 397-398
- share-nothing architecture, 392-393
- share-something architecture, 393-395
- sharing data, 392-398
- shell, exiting, 289
- Short Message Service Center (SMSC), 432
- ShutdownTime element (child specification), 183
- shutdown_time flag, 294, 302
- signal event, 247
- simple target systems, 265
- simple_one_for_one strategy, 181, 189-191, 195
- single point of failure, 387-388, 407
- site redundancy, 387
- sleep() function, 89, 92
- sloppy quorums, 369
- SMP (symmetric multiprocessing), 13
- SMSC (Short Message Service Center), 432
- sname flag, 294, 300
- snmp agent, 8
- snmp client, 8
- SOA (service-oriented architecture), 12, 375
- soak testing, 412
- soft real-time, 3
- software upgrades
 - about, 318-320
 - adding states, 323-326
 - coffee machine FSM example, 320-323
- soft_purge() function, 45
- spawn() function
 - erlang module, 29, 49
 - proc_lib module, 244
- spawning processes
 - about, 29, 54
 - spawn options, 108-114
 - supervision trees and, 170
- spawn_link() function, 35, 49
- spawn_opt() function, 108-114
- special processes
 - about, 16, 241
 - asynchronously starting, 246
 - dynamic modules and hibernating, 255
 - entire mutex example, 251-255
 - handling exits, 247-249
 - mutex states and, 247
 - mutexes and, 242-244
 - proc_lib module and, 195
 - starting, 244-246
 - system messages, 249
 - trace and log events, 250
 - upgrading, 350
- spike testing, 412
- spiral (metrics), 435
- src directory, 206, 208
- SSL, 366
- ssl module, 373-375
- St. Laurent, Simon, 14
- stack overflow, 23
- start phases (applications)
 - about, 226-228
 - included applications and, 228-230
- start script, 267, 273
- start() function
 - application module, 165, 210, 221, 226
 - gen_event module, 149
 - gen_fsm module, 129
 - gen_server module, 99
 - observer module, 217, 295
 - proc_lib module, 244
 - rb module, 239
 - supervisor module, 176
- start.boot file, 282
- StartFunction element (child specification), 183
- start_boot() function, 280



- start_child() function, 187, 337
- start_clean.boot file, 282
- start_ertl program, 267, 288
- start_link() function
 - gen_event module, 149
 - gen_fsm module, 127-130
 - gen_server module, 78, 80-82
 - proc_lib module, 244
 - supervisor module, 176, 210
 - supervisor_bridge module, 194
- start_sasl.boot file, 282
- state-based alarms, 437
- states (see FSMs)
- static clusters, 366
- statistics() function, 105, 108
- stdlib application
 - about, 7, 205
 - alternative boot files, 282
 - application resource files and, 214
 - release resource files and, 269
- stop() function
 - application module, 211, 222
 - gen_event module, 149, 152
 - init module, 302
- stress testing, 412
- strict quorums, 369
- supervision trees
 - about, 10, 170-175
 - multiple policies and, 197
 - special processes and, 242
 - starting the supervisor, 176-179
 - supervisor bridges and, 194-195
 - supervisor specification, 179-186
 - testing strategy, 199-199
- supervisor bridges, 194-195
- supervisor module
 - about, 16, 175
 - dynamic children, 186-193
 - starting the supervisor, 176-179
- supervisor reports (SASL), 238
- supervisor specification
 - about, 179
 - child specification, 171, 183-193
 - restart specification, 179-183
- supervisor:check_childspecs() function, 186
- supervisor:count_children() function, 185
- supervisor:delete_child() function, 189, 190, 337
- supervisor:restart_child() function, 190
- supervisor:start() function, 176
- supervisor:start_child() function, 187, 337
- supervisor:start_link() function, 176, 210
- supervisor:terminate_child() function, 190, 337
- supervisor:which_children() function, 185, 189
- supervisors
 - about, 10, 11, 170-175
 - application master and, 205
 - combining with applications, 230
 - dynamic children and, 186-193
 - handling errors, 158-160
 - links and monitors for, 34-38
 - scalability and short-lived processes, 195-197
 - starting, 171, 176-179
 - supervisor specification, 179-186
 - synchronous starts, 197
 - terminating, 171
- supervisor_bridge:start_link() function, 194
- support automation, 439-440
- suspend() function, 106, 108, 338
- swap_handler() function, 160
- swap_sup_handler() function, 161
- symmetric multiprocessing (SMP), 13
- synchronous events
 - sending, 131, 153-156
 - sending to generic FSM, 138-139
- synchronous message passing, 82-83, 88
- sync_event() function, 156
- sync_nodes low-level instruction, 351
- sync_notify() function, 153
- sync_send_all_state_event() function, 138-139
- sync_send_event() function, 138-140
- syntax_tools modules, 8
- sys module
 - about, 101
 - debug options, 195
 - handling system messages, 249
 - implementing trace functions, 103-105
 - statistics, status, state, 105-107
 - system messages, 103
 - tracing and logging, 101-102
 - user-defined behaviors and, 256
- sys:change_code() function, 348
- sys:debug_options() function, 245, 249
- sys:get_state() function, 106, 108
- sys:get_status() function, 105, 108
- sys:handle_debug() function, 250, 251
- sys:handle_system_message() function, 249



- `sys:install()` function, 104, 108, 251
- `sys:log()` function, 102, 108, 250
- `sys:log_to_file()` function, 102, 108
- `sys:remove()` function, 105, 108
- `sys:replace_state()` function, 107, 108
- `sys:resume()` function, 106, 108
- `sys:statistics()` function, 105, 108
- `sys:suspend()` function, 106, 108, 338
- `sys:trace()` function, 102, 108, 250
- syslog tool, 430
- System Application Support Libraries (see `sasl` application)
- system blueprints, 363, 419
- system messages, 249, 373
- system principles
 - about, 264-265
 - design principles, 10-11
 - release handling, 265-303
- SystemTap probe, 9
- `system_continue()` callback function, 250, 253
- `system_info()` function, 272, 291
- system_information module, 270
- `system_monitor()` function, 373, 417
- `system_terminate()` callback function, 250, 253
- systools module
 - about, 273
 - appup files and, 333
 - creating boot file, 274-277
- `systools:make_relup()` function, 340
- `systools:make_script()` function, 274-277, 281, 300, 339
- `systools:make_tar()` function, 284, 340, 346
- `systools:script2boot()` function, 282
- s_groups, 372

T

- +t emulator flag, 291
- tail recursive functions, 23, 118
- takeovers, 222
- target systems
 - basic, 264
 - embedded, 283, 293
 - simple, 265
- TCP stream example, 256-260
- temporary application type, 221, 271
- `terminate()` callback function
 - gen_event and, 152, 154
 - gen_fsm and, 140
 - gen_server and, 78, 79, 89-91
 - special processes and, 248
 - supervisor and, 183, 193
 - supervisor_bridge and, 195
- `terminate_child()` function, 190, 337
- terminating
 - applications, 221
 - generic FSMs, 139-140
 - generic servers, 89-91
 - processes, 36, 54, 158-160, 173
 - supervisors, 171
- Test Server framework, 9
- testing capacity, 412-413
- test_server application, 9
- text messages, 432
- threshold-based alarms, 437
- throughput, 410, 413, 414
- throw exception, 34
- time (metrics), 435
- timeouts
 - behavior, 114
 - call, 91-95
 - generic FSMs, 130, 135-136
 - generic servers, 95-97
- timer:apply_after() function, 97
- timer:apply_interval() function, 97
- timer:send_after() function, 97
- timer:send_interval() function, 97
- timer:sleep() function, 89, 92
- timestamps (metrics), 435
- tools and libraries, 7-9
- tools application, 9
- to_eri command, 267, 289
- trace() function, 102, 108, 250
- transform_table() function, 333
- transient application type, 222, 271
- trap_exit flag, 37, 54, 245
- try-catch expression, 34, 54
- tv (table visualizer), 218
- two-module version limit, 318
- typer application, 267

U

- Udon web server, 371
- undef runtime error, 151, 322
- unhandled messages, 86-88
- unique sequence numbers, 390
- units of time, 435
- University of Glasgow, 372
- unlink() function, 35



unpack_release() function, 344, 347

unregister_name() function, 98

update_counter() function, 436

upgrades

- emulator and core applications, 352

- environment variables, 350

- in distributed environments, 351

- module, 43-45

- release, 326-350

- software, 318-326

- special processes, 350

- to records, 333

- with rebar3 tool, 353-355

V

variables

- environment, 219-221, 222, 268, 350

- mutable, 24

- pattern matching and, 31

versions

- release and application, 272

- software, 318

- two-module limit, 318

vertical scaling, 405-407

Virding, Robert, 117

virtual binary heap, 110

vnodes, 368

-vsn module attribute, 79, 319

W

wait event, 247

werl program, 267

whereis() function, 165

whereis_name() function, 98

which_applications() function, 212, 213

which_children() function, 185, 189

which_releases() function, 349

Wiger, Ulf, 138, 376, 422

Wikström, Claes "Klacke", 299

wildcard() function, 40

Williams, Mike, 117

worker processes, 10

wx graphics package, 8

X

xmerl (XML parser), 8

Y

Yaws web server, 203, 204, 299

yecc parse generator, 9

Z

Zookeeper Atomic Broadcast (ZAB), 396



关于作者

Francesco Cesarini 是 Erlang Solutions 公司 (<http://www.erlang-solutions.com/>) 的创始人与技术总监。早在 1995 年,他便在爱立信计算机科学实验室——Erlang 的诞生地——开始了实习生涯,自那开始至今,他每天都在使用 Erlang 编程语言。之后他转到了爱立信的 Erlang 培训与咨询部门,参与开发了 OTP 的 R1 版本,致力于提供顶尖级电信应用的一站式解决方案。1999 年, Erlang 转为开源方式发布后不久,他便开始创业,最终发展成如今我们所见的 Erlang Solutions 公司。该公司跨越三大洲,在七个国家设有办事处,已成为高伸缩性、高可用性的端到端解决方案领域的首选合作伙伴。作为技术总监, Francesco 领导 Erlang Solutions 公司的开发和咨询团队,负责产品和战略研究。他同时还是 *Erlang Programming* 一书的合著者,该书由 O'Reilly 出版社出版。他在哥德堡 IT 大学 (IT University of Gothenburg) 讲课已超过十年,并且自 2010 年起开始在牛津大学教授面向并发的编程 (concurrency-oriented programming) 课程。你可以通过搜索 @FrancescoC 在 Twitter 上找到他。

Steve Vinoski 是 Arista Networks 公司的软件开发人员。在他超过 30 年的软件开发生涯中,大部分时间都在从事中间件与分布式计算系统的研发。直到 2006 年,在他已经用 C++ 和 Java 开发中间件系统 20 年后,他发现了 Erlang,自那以后,他便开始把 Erlang 作为自己的主力开发语言。Steve 曾经为各种各样的 Erlang 项目做过贡献,包括 Riak 数据库与 Yaws Web 服务器等。他还为 Erlang/OTP 代码库贡献过许多的功能修正与增强补丁。

Steve 也是一位长期作家,曾独自撰写或与人合著过许多涉及中间件、分布式系统、Web 开发等方面的文章、论文和书籍,总数量过百。2008 年至 2012 年期间,他为 *IEEE Internet Computing (IC)* 杂志撰写 *The Functional Web* (函数式风格的 Web) 专栏;在此之前的 2002 年至 2008 年间则为 *IC* 撰写 *Toward Integration* (迈向集成) 专栏。他还是该杂志的编辑委员会成员。从 1995 年到 2005 年, Steve 与其他人一起为 *C++ Report* 以及之后的 *C/C++ User Journal* 撰写 *Object Interconnections* (对象互联) 专栏,该专栏颇受欢迎,主要涉及分布式对象计算相关的主题。多年来, Steve 参加了许多与中间件、分布式系统、Web 开发、编程语言等主题相关的会议,进行演讲和授课达数百次,并担任了许多会议和研讨会的主席或计划委员会成员。



封面介绍

本书封面上的动物是一种被称为欧洲鳎（European plaice，学名 *Pleuronectes platessa*）的常见比目鱼。欧洲鳎生活在欧洲沿岸，北至巴伦支海，南至地中海。在水深 10 至 50 米的范围内能找到它们，它们会在水底的沉积物里挖洞。

欧洲鳎拥有深绿色或棕色的鳞片，并附着有橙色的斑点。成年个体可以长到 1 米，但大多数只有半米左右。它们以海洋蠕虫、双壳类和甲壳类动物为食。

欧洲鳎是北德和丹麦日常烹饪的常见食材，通常是在欧洲被渔民捕获的。在丹麦，配上炸薯条和奶油蛋卷酱的煎炸欧洲鳎特别受欢迎。20 世纪 70 年代，欧洲渔业曾过度捕捞欧洲鳎，但是如今由于环境保护得力，其数量正在不断增加，并且在 2012 年达到了自 1957 年以来的最高水平。

O'Reilly 出版的图书封面上的动物，很多都是处于濒临灭绝的动物，它们存在的重要性不言而喻。要了解更多有关如何救助这些濒危动物的知识，请访问 animals.oreilly.com。

高伸缩性系统: Erlang/OTP大型分布式容错设计

如果你需要设计一套规模可伸缩且具有高可用性的容错系统,那么 Erlang/OTP 平台值得你去深入了解,因为其适用领域广阔、技术积累深厚,具备丰富功能的同时又贯彻了高度一致的设计思想。这本实践指南展示了如何使用 Erlang 编程语言及其 OTP 框架(其中包含许多可复用的库、工具和设计原则等),你将可以基于简单的理念开发出复杂的商业级别的系统,并使系统具备故障免疫能力。

从本书的第1部分中,你将学会如何基于 Erlang/OTP 设计和实现进程行为模式与监督树,以及将它们打包成独立节点的方法。在本书的第2部分中,你将了解到贯穿整个系统的可靠性、规模可伸缩性和高可用性方面的设计知识。如果你熟悉 Erlang,本书将会帮助你理解那些为了使系统能够持续运转所需要做出的设计决策以及权衡。

- 探索OTP的基石: Erlang编程语言、相关工具、可复用的库集合,以及相关抽象理念与设计规则。
- 深入OTP实现具有可复用性的核心机制: 各类行为模式内部涉及的 Erlang 进程结构。
- 理解OTP中行为模式是如何为客户端-服务器结构、有限状态机模式、事件处理、运行时、代码集成等功能提供支持的。
- 编写自己的行为模式以及特殊进程。
- 使用OTP提供的工具、技术与架构来处理部署、监视和运维等。

“值得每一位Erlang/Elixir开发者阅读。”

——Saša Juric
Elixir in Action作者

Francesco Cesarini, Erlang Solutions公司的创始人与技术总监。自1995年起便使用 Erlang,并且是 Erlang R1 版本开发团队的一员。他致力于教授开发人员、DevOps工程师、测试人员、项目管理人员以及大学生等掌握 Erlang/OTP。

Steve Vinoski, Arista Networks公司的软件工程师。在分布式系统、编程语言、系统集成以及服务端 Web 开发领域拥有极其丰富的经验。他向 Erlang/OTP 以及其他 Erlang 项目贡献过大量的补丁和功能。

PROGRAMMING/ERLANG

图书分类: 编程语言/Erlang

策划编辑: 张春雨

责任编辑: 刘舫



Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc. 授权电子工业出版社出版

此简体中文版仅限于中国大陆(不包含中国香港、澳门特别行政区和中国台湾地区)销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-33747-5



9 787121 337475 >

定价: 115.00元